Evolutionary algorithms

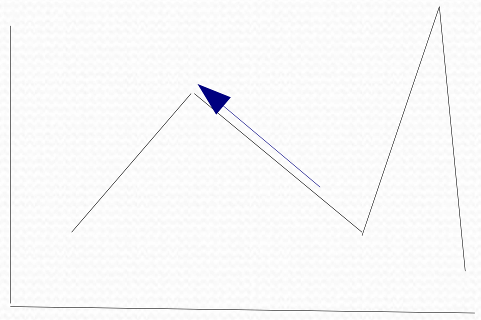
- Simple genetic algorithms
- Evolutionary Strategies
- Genetic Programming

Partially based on slides by Thomas Bäck

Heuristic Search

- SAT solvers, CP solvers, ILP solvers:
  - find exact solutions to discrete constraint optimization problems
  - can be time consuming
- Heuristic solvers:
  - employ "heuristics": guidelines for finding good solutions quickly
  - don't find exact solutions
  - can be much faster
  - can deal with problems that are numerical and not in a "nice" form (eg., linear)

Hill-Climbing



- Hill-climbing is arguably the simplest heuristic algorithm

```
1. $S$ = arbitrary candidate solution
2. $S'$ = solutions in the neighborhood of $S$
3. **if** best solution in $S'$ is not better than $S$ **then** stop
4. let $S$ be the best solution in $S'$
5. go to 2.
```

Neighborhood Search

- Important choice in hill-climbing: which neighborhoods to consider
  - Add a small value to each coordinate? Substruct a small value from each coordinate?

$$\begin{tikzcd}
 & (x_1 + \epsilon, x_2, \ldots, x_n) & {} \\
(x_1, x_2, \ldots, x_n) \arrow[ur] \arrow[r] \arrow[dr] \arrow[ddr] & (x_1 - \epsilon, x_2, \ldots, x_n) \arrow[ur] \arrow[r] \arrow[dr] \arrow[ddr] & {} \\
 & (x_1, x_2 + \epsilon, \ldots, x_n) & {} \\
 & (x_1, x_2 - \epsilon, \ldots, x_n) & {} \\
 & \vdots &
\end{tikzcd}$$

Large Neighborhood Search

- Iteratively select a random subset of variables of limited size, find an optimal assignment for these variables, assuming the others are fixed
  - Requires the availability of an algorithm to solve the intermediate problems optimally (linear programming, CP, ..)

$$\begin{tikzcd}
(x_1, x_2, \ldots, x_n) \arrow[d] \\
(?, ?, x_3, \ldots, x_n) \arrow[d] \\
(x'_1, x'_2, x_3, \ldots, x_n) \arrow[d] \\
(x'_1, x'_2, ?, x_4, \ldots, x_n) \arrow[d] \\
(x'_1, x'_2, x'_3, x_4, \ldots, x_n)
\end{tikzcd}$$

Other Well-known Heuristic Search Strategies

- Simulated annealing
- Tabu search
- Evolutionary algorithms
  - genetic algorithms
  - genetic programming
  - evolutionary strategies
- Artificial ants
- Particle swarms

Advantages of GAs

- Evolution and natural selection has proven to be a robust method
- A “black box” approach that can easily be applied to many optimization problems
- GAs can be easily parallelized and run on multiple machines

Some definitions

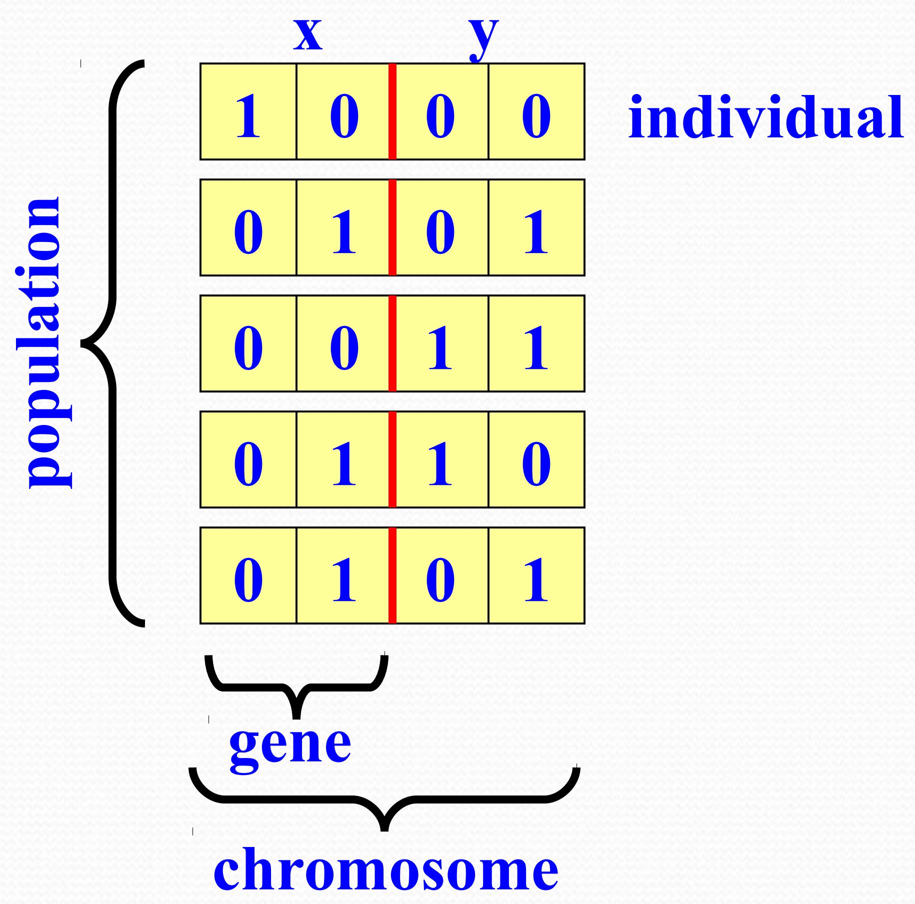
- **Population**: a collection of solutions for the studied (optimization) problem
- **Individual**: a single solution in a GA
- **Chromosome (genotype)**: representation for a single solution
- **Gene**: part of a chromosome, usually representing a variable as part of the solution

Some definitions

- **Encoding**: conversion of a solution to its equivalent representation (chromosome)
- **Decoding**: conversion of a chromosome (**genotype**) to its equivalent solution (phenotype)
- **Fitness**: scalar value denoting the suitability of a solution

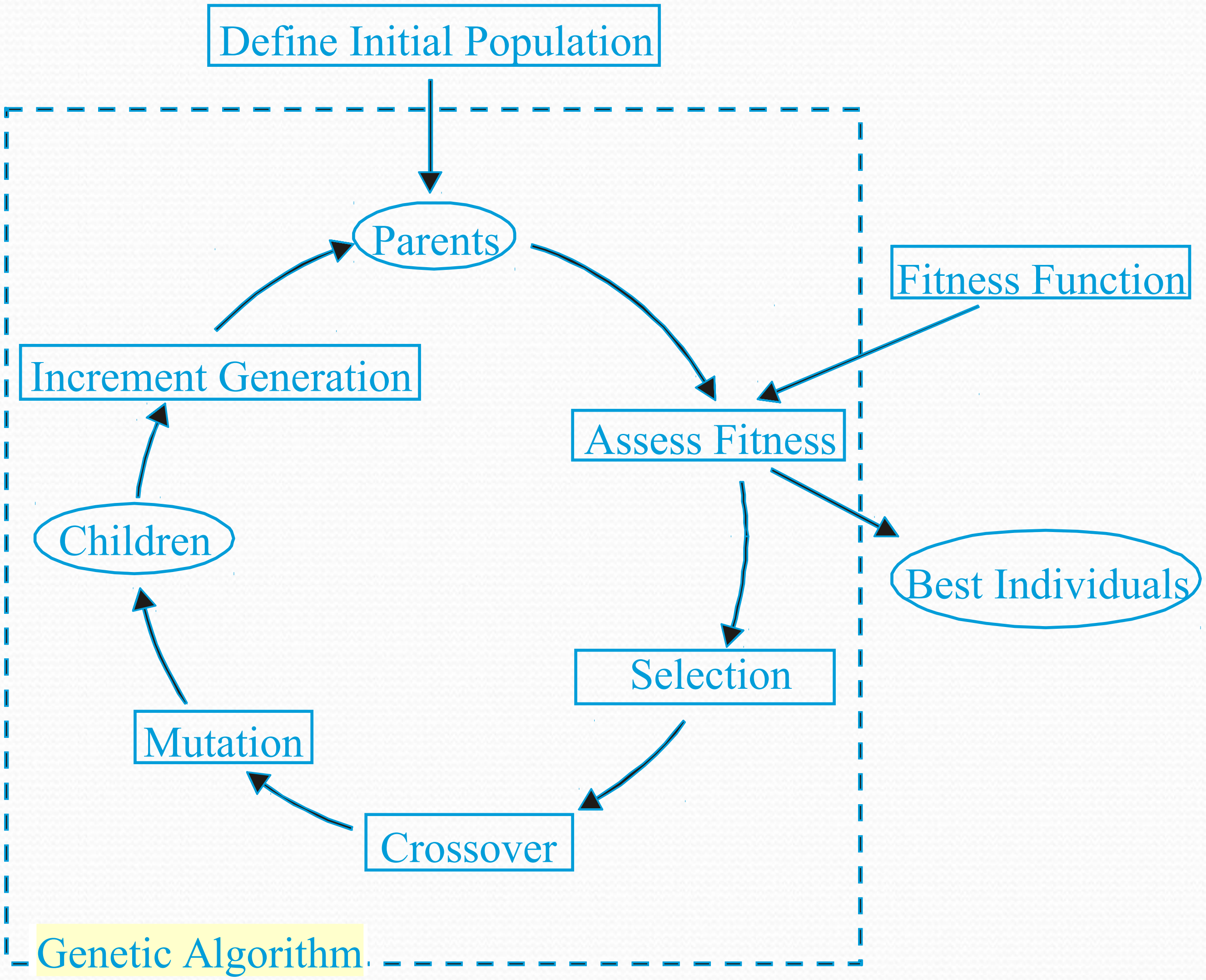
GA terminology

Generation t



solution	fitness
(2,0)	4
(1,1)	2
(0,3)	3
(1,2)	3
(1,1)	2

Genetic algorithm



Pseudo code

- Initialize population *P*:
  - E.g. generate random *p* solutions
- Evaluate solutions in *P*:
  - determine for all $h \in P$, Fitness(h)
- **While** terminate is FALSE
  - Generate new generation *P* using genetic operators
  - Evaluate solutions in *P*
- **Return** solution $h \in P$ with the highest Fitness

Termination criteria

- Number of generations
  (restart GA if best solution is not satisfactory)
- Fitness of best individual
- Average fitness of population
- Difference of best fitness (across generations)
- Difference of average fitness (across generations)

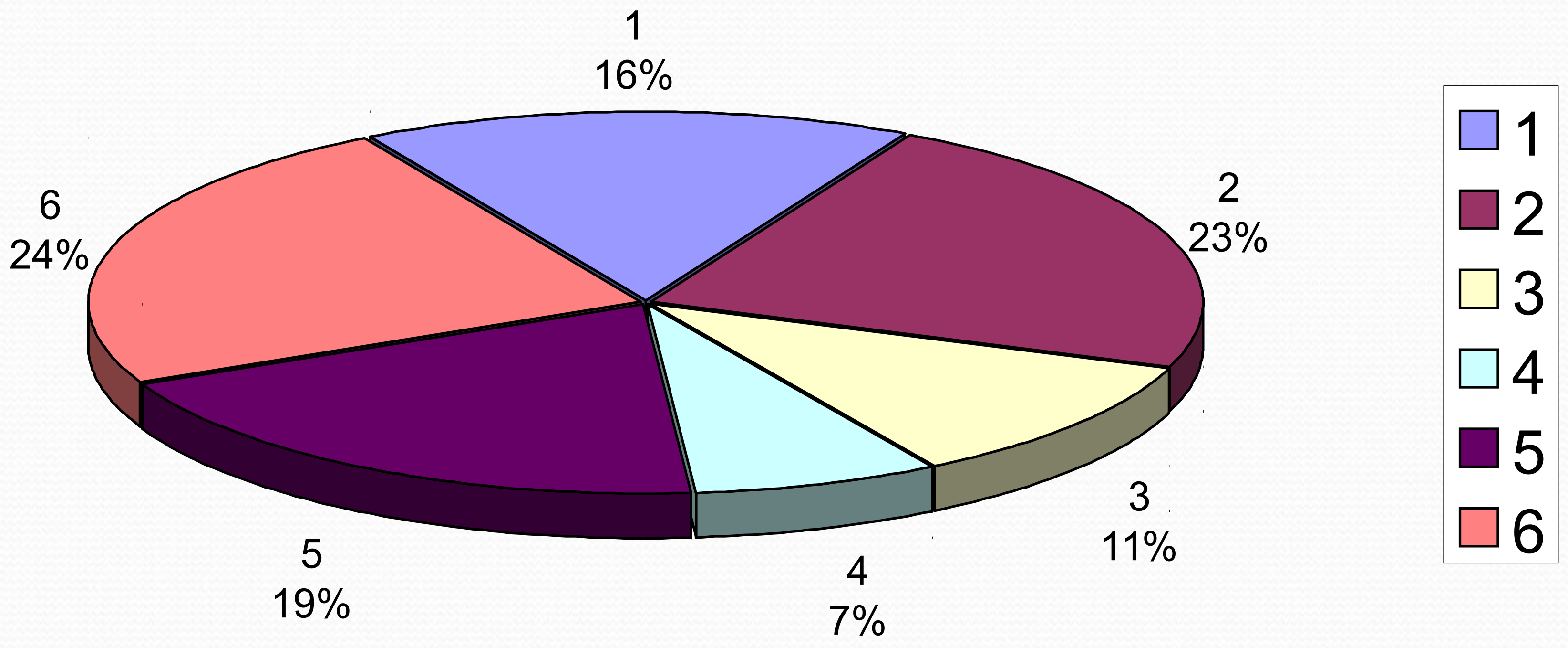
Reproduction

Three steps:

- Selection
- Crossover
- Mutation

In GAs, the population size is often kept constant. The programmer is free to choose which methods to use for all three steps.

Roulette-wheel selection



Roulette-wheel selection

$$
\begin{array}{r}
 \\
p = 0.16 \\
p = 0.23 \\
p = 0.11 \\
p = 0.07 \\
p = 0.19 \\
p = 0.24
\end{array}
\begin{array}{c}
\text{individuals} \\
\begin{bmatrix} 01100 \\ 10001 \\ 11010 \\ 00111 \\ 11000 \\ 10110 \end{bmatrix}
\end{array}
\begin{array}{c}
\text{fitness} \\
\begin{bmatrix} 34 \\ 48 \\ 23 \\ 15 \\ 41 \\ 50 \end{bmatrix}
\end{array}
\xrightarrow{\text{selection}}
\begin{bmatrix} 01100 \\ 10001 \\ 10001 \\ 11000 \\ 10110 \\ 10110 \end{bmatrix}
$$

Sum = 211

Cumulative probability: 0.16, 0.39, 0.50, 0.57, 0.76, 1.00

Tournament selection

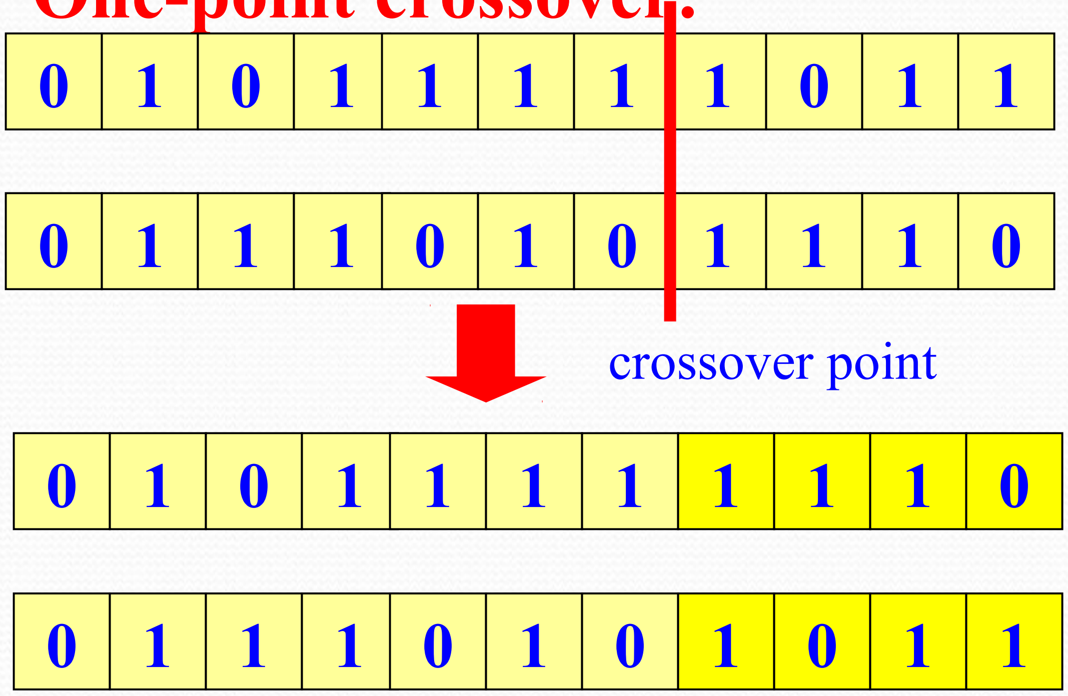
- Select pairs randomly
- Fitter individual wins
  - deterministic
  - probabilistic
    - constant probability that the better individual wins
    - probability of winning depends on fitness

Tournament selection can also be combined with roulette-wheel selection.

Crossover

- Exchange parts of chromosome with a crossover probability (p_c is usually about 0.8)
  - i.e., with probability 1-p_c no crossover takes place
- Select crossover points randomly

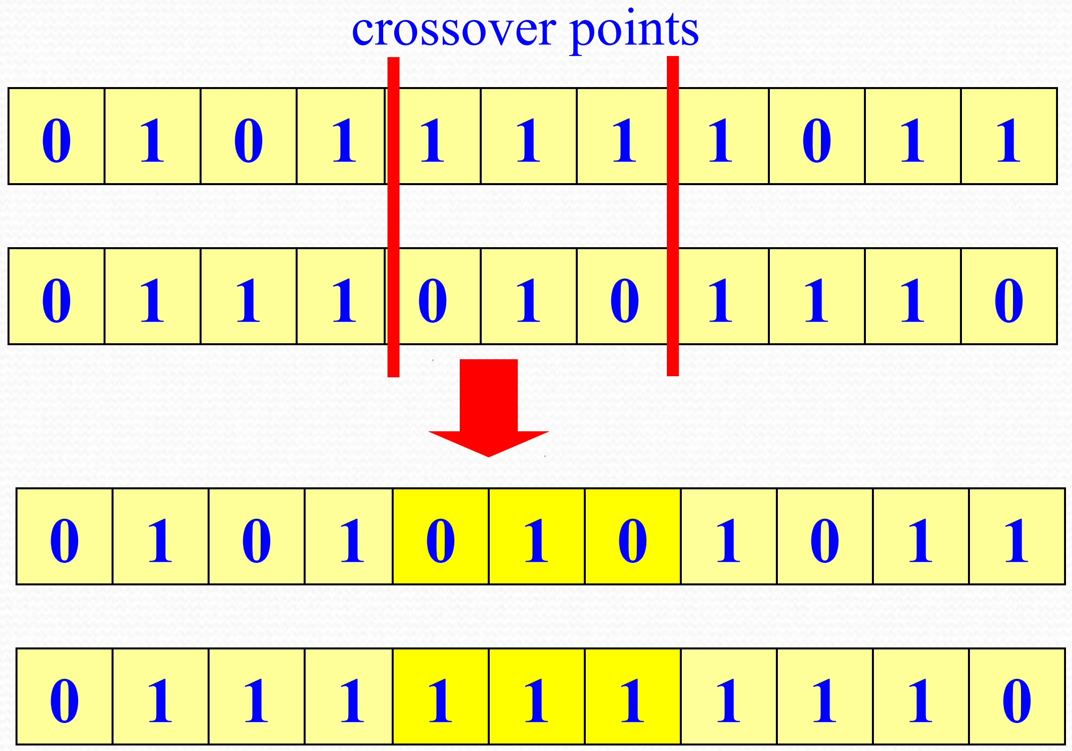
One-point crossover:



N-point crossover

- Select N points for exchanging parts
- Exchange multiple parts

Two-point crossover:



Uniform crossover

- Exchange bits using a randomly generated mask

0	1	0	1	0	0	1	0	0	1	1	mask

0	1	0	1	1	1	1	1	0	1	1
0	1	1	1	0	1	0	1	1	1	0

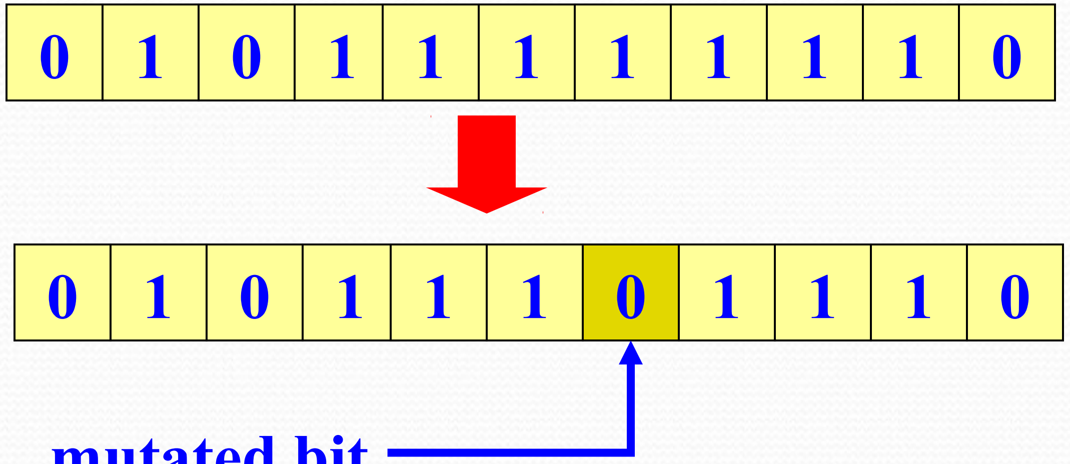
$\Downarrow$

0	1	0	1	1	1	0	1	0	1	0
0	1	1	1	0	1	1	1	1	1	1

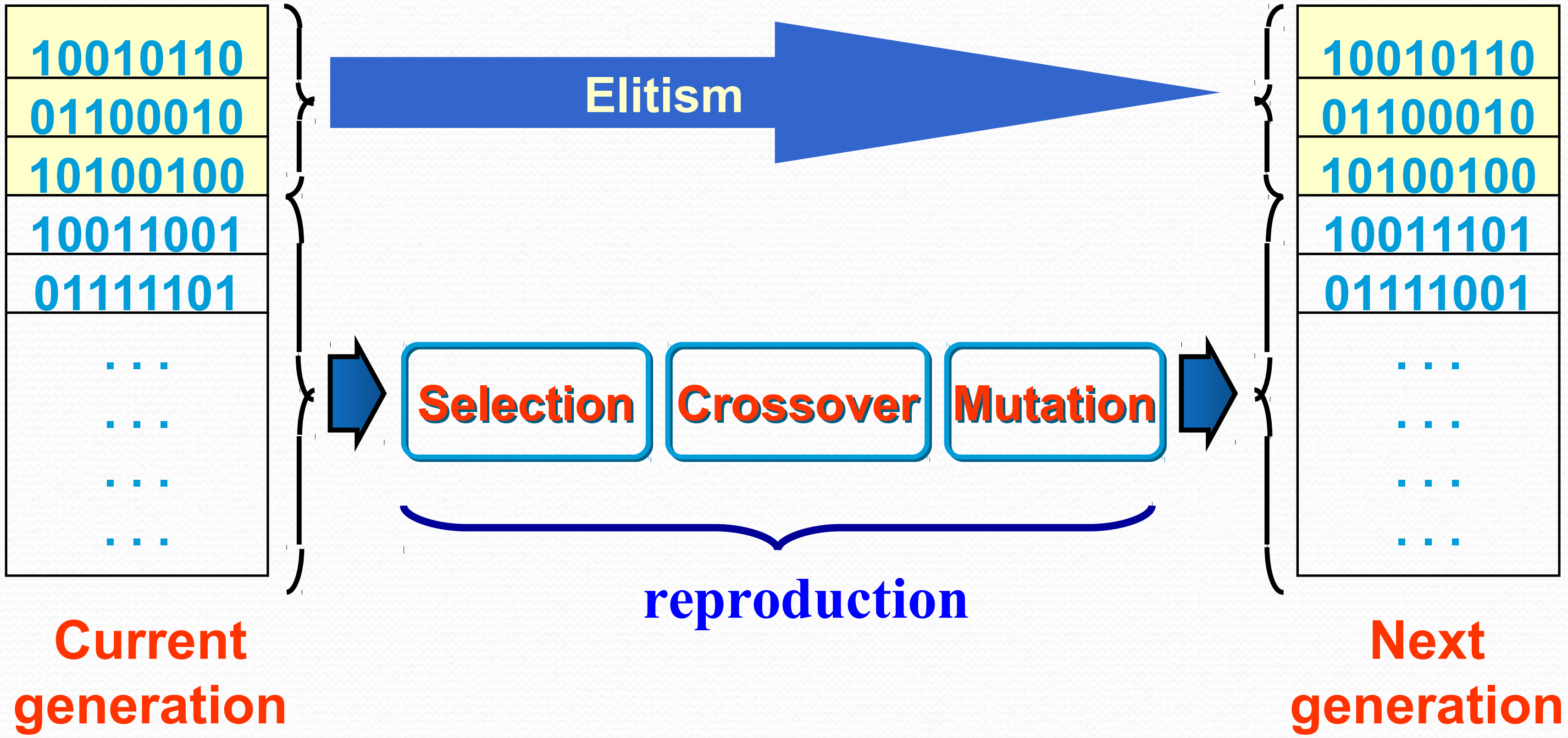
Mutation

- Crossover is used to search the solution space
- Mutation is needed to escape from local optima
- Introduces genetic diversity
- Mutation is rare (p_m is about 0.005)

Uniform mutation:



GA iteration



Encoding and decoding

- Common coding methods
  - “standard” binary integer coding
  - Gray coding (binary)
  - real valued coding (*evolutionary strategies*)
  - tree structures (*genetic programming*)

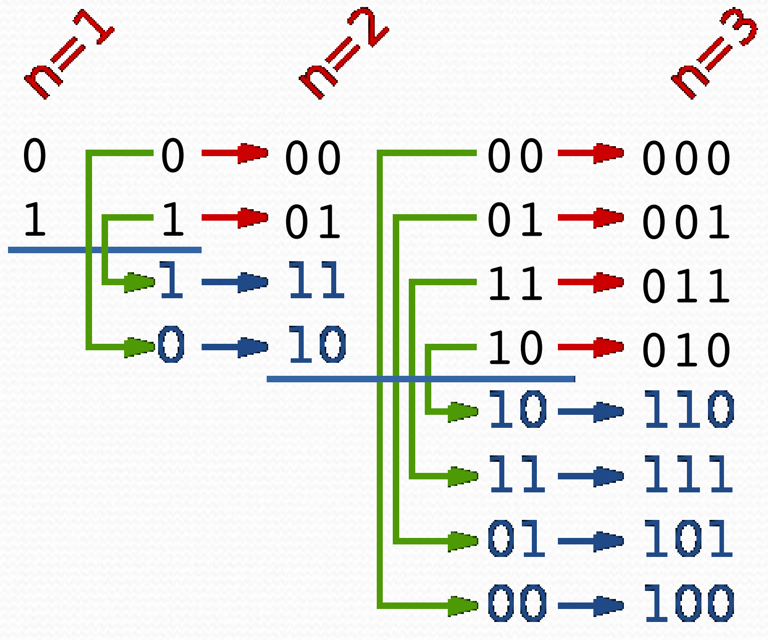
Gray Coding

- Aim: binary coding of integers such that integers *x* and *y* for which $|x-y|=1$ only differ in one bit

Dec	Gray	Binary
0	000	000
1	001	001
2	011	010
3	010	011
4	110	100
5	111	101
6	101	110
7	100	111

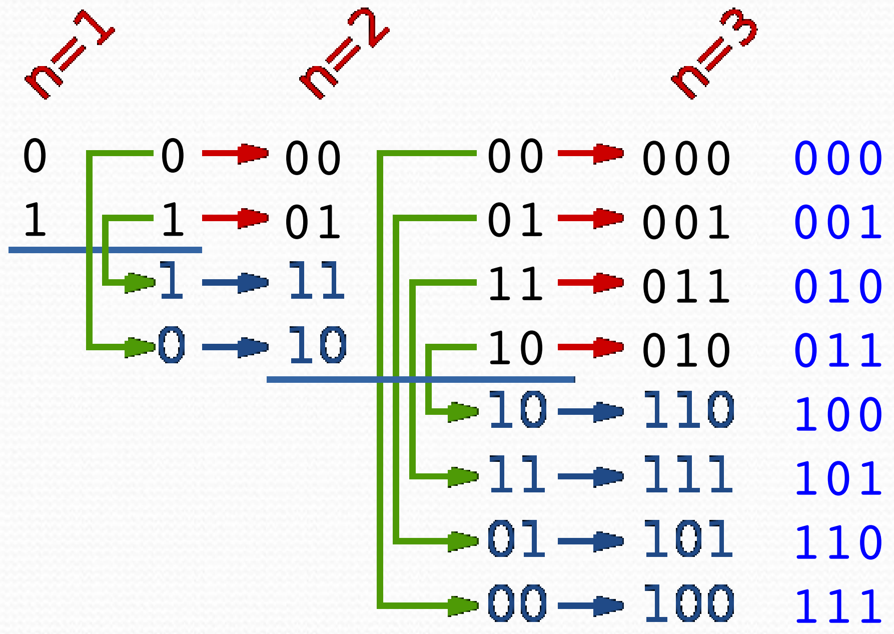
Gray Coding

- Codes for *n*=1: (i.e., integers 0, 1)
  0   1
- Codes for *n*=2: (i.e., integers 0, 1, 2, 3)
  Reflected entries for *n*=0:
  1   0
  Prefix old entries with 0:
  $\underline{0}0$ $\underline{0}1$
  Prefix reflected entries with 1:
  $\underline{1}1$ $\underline{1}0$
  Codes hence:
  $\underline{0}0$ $\underline{0}1$ $\underline{1}1$ $\underline{1}0$
- Codes for *n*=3: (i.e., integers 0, 1, 2, …, 7)
  Reflected entries for *n*=2:
  10   11   01   00
  Codes hence:
  $\underline{0}00$ $\underline{0}01$ $\underline{0}11$ $\underline{0}10$ $\underline{1}10$ $\underline{1}11$ $\underline{1}01$ $\underline{1}00$



Gray Coding

- Given a “normal” bit representation, how to calculate the Gray code?



bitstring	Gray
10100	11110
10101	11111
10110	11101
11001	10101

A bit flips in the Gray code iff the bit before it has value 1 in the original code.

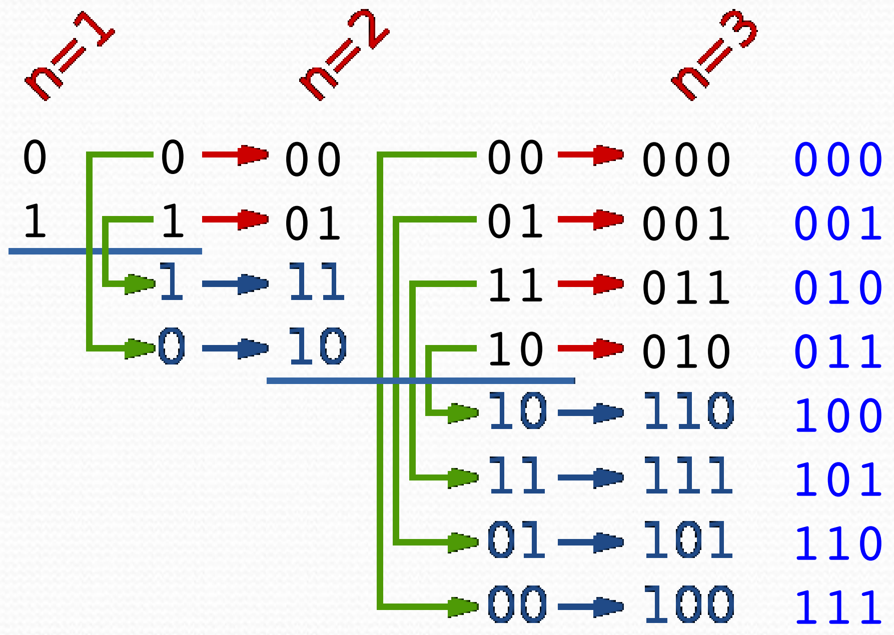
Gray Coding

- Source code in Python for calculating Gray code:

```
def binaryToGray(num):
        return (num >> 1) ^ num
```

Gray Coding

- Given a Gray code, how to calculate a “normal” bit representation?



bitstring	→	Gray
10100	→	11110
10101	→	11111
10110	→	11101
11001	→	10101

A bit flips in the “normal” code (as compared to the Gray code) iff the bit before it has value 1 in the “normal” code.

Gray Coding

- Gray coding does not avoid that integers far away from each other can have similar codes

  00000=0

  10000=31

  → Mutation can still change numbers a lot

- Gray coding only ensures that there always is a one-bit mutation to transform integer *x* into integer *x+1* or *x-1*.

Constraints

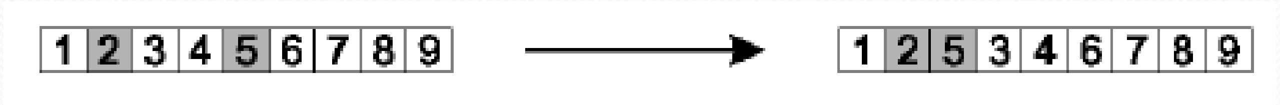
- Examples:
  - “A string of numbers should represent a permutation”
    (1,2,3) is valid; (1,1,3) is not
  - “The sum of numbers should not be lower than a threshold”
- Possibility 1: fitness function modification
  - setting fitness of unfeasible solutions to zero
    (search may be very inefficient due to unfeasible solutions)
  - penalty function (negative terms for violated constraints)
  - barrier function (already penalty if “close to” violation)

Constraints

- Possibility 2 (preferred method): special encoding
    - GA searches always through allowed solutions
    - smaller search space
    - ad hoc method, may be difficult to find
- Example: permutations (see AI course)

Mutations for Permutations

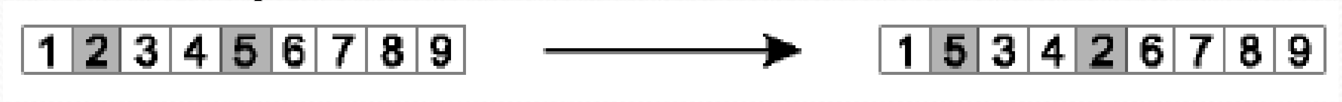
- Insert mutation:
  - Pick two allele values at random
  - Move the second to follow the first, shifting the rest along to accommodate
  - Note: this preserves most of the order and adjacency information; changes the position of numbers a lot



Removed Adjancency:	(2,3), (4,5), (5,6)
Added Adjacency:	(2,5), (4,6), (5,3)
Removed orders:	3->5, 4->5
Added orders:	5->3, 5->4
Changed positions:	3, 4, 5

Mutations for Permutations

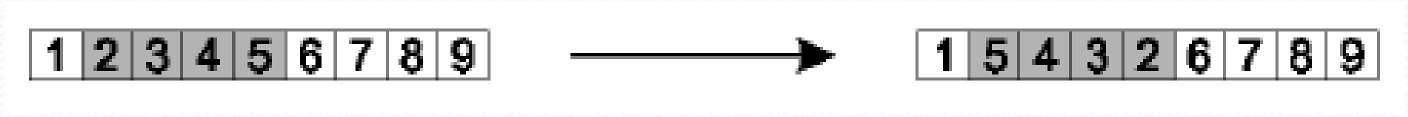
- Swap mutation:
  - Pick two alleles at random and swap their positions
  - Disrupts adjacency information and order more; preserves positions



Removed Adjancency:	(1,2), (2,3), (4,5), (5,6)
Added Adjacency:	(1,5), (2,6), (4,2), (5,3)
Removed order:	2->3, 2->4, 2->5, 3->5, 4->5
Added order:	5->3, 5->4, 3->2, 4->2, 5->2
Changed positions:	2, 5

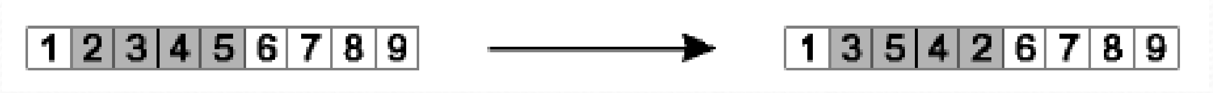
Mutations for Permutations

- Inversion mutation:
  - Pick two alleles at random and then invert the substring between them.
  - Preserves most adjacency information (only breaks two links) but disruptive for order information



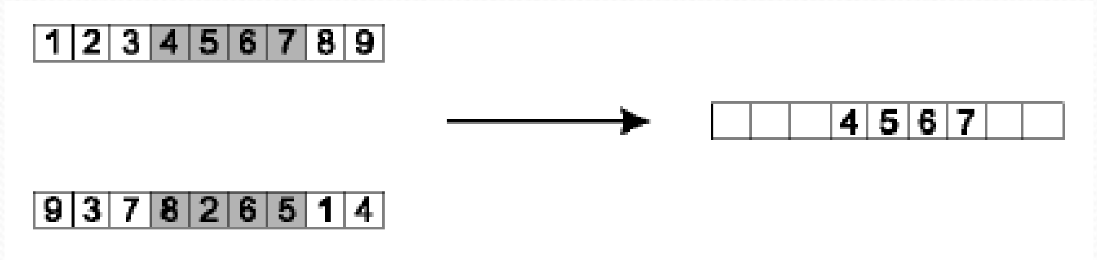
Mutations for Permutations

- Scramble mutation:
  - Pick a subset of genes at random (not necessarily consecutive)
  - Randomly rearrange the alleles in those positions

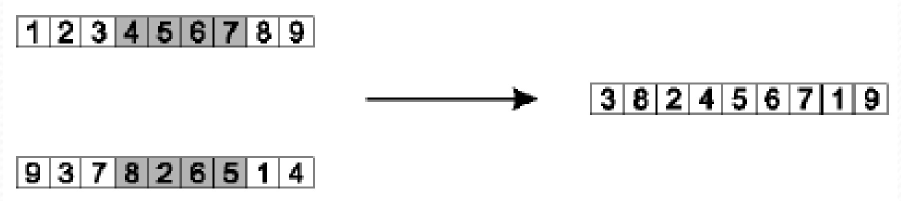


Crossover for Permutations

- Order one crossover:
  - Choose an arbitrary part from the first parent, copy this part to the first child



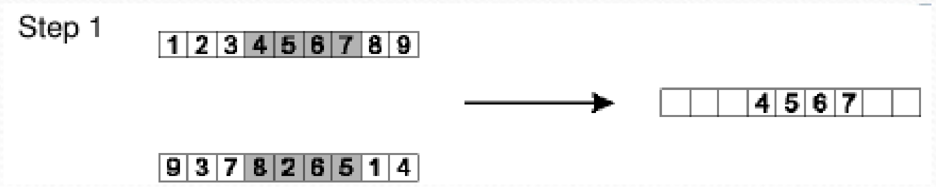
  - Copy the numbers that are not in the first part, to the first child:
    - starting right from cut point of the copied part,
    - using the order of the second parent and wrapping around at the end



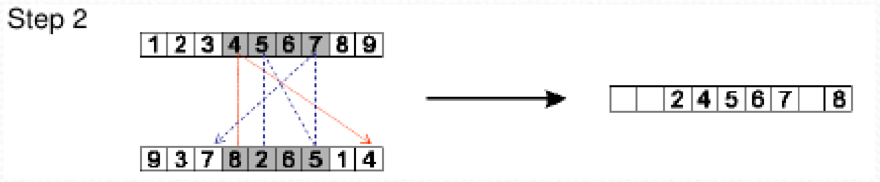
  - Analogous for the second child, with parent roles reversed

Crossover for Permutations

- Partially Mapped Crossover (PMX):
  - Choose random segment and copy it from P1

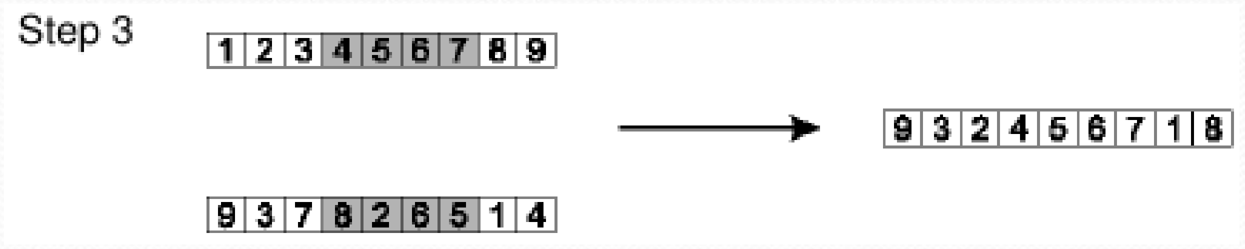


  - Starting from the first crossover point look for elements in that segment of P2 that have not been copied
  - For each of these i look in the offspring to see what element j has been copied in its place from P1
  - Place i into the position occupied j in P2, since we know that we will not be putting j there (as is already in offspring)
  - If the place occupied by j in P2 has already been filled in the offspring k, put i in the position occupied by k in P2



Crossover for Permutations

- Partially Mapped Crossover (PMX):
  - Having dealt with the elements from the crossover segment, the rest of the offspring can be filled from P2.



- Idea: <u>maintain position</u>

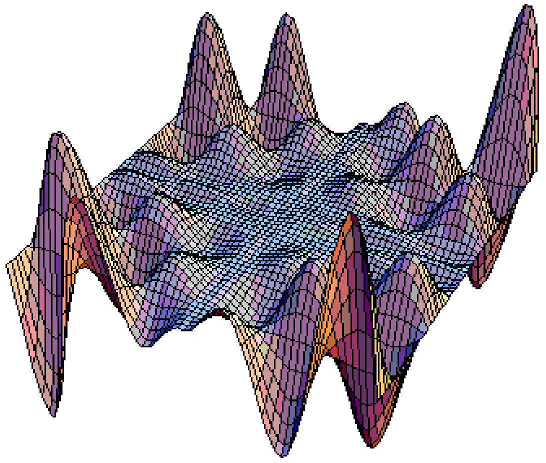
Order vs Position in Permutations

- Order, but not position of numbers is important in problems such as the traveling salesman problem (visiting all cities in a certain order)

- Position, but not order of numbers is important in problems such as allocating visitors in hotels to rooms (visitors have to be allocated once to one room, but the order of the allocation does not matter)

Evolutionary Strategies

- Numerical optimization problems:
  - **Given** a function *f* from real numbers to a real number
  - **Find** coordinates at which *f* is maximized



Evolutionary Strategies

- Main idea:
  individuals consist of vectors of real numbers (not binary)
- Redefinitions of
  - selection
  - crossover
  - mutation
- Operations executed in the order
  crossover → mutation → selection

ES: Selection

- Not performed *before* mutation and crossover, but *after* these operations
- It is assumed mutation (& crossover) generate $\lambda > \mu$ individuals (where μ is population size) (typically $\lambda \approx 7\mu$)
- Deterministically eliminate worst individuals from
  - children only: (μ,λ)-ES → escapes from local optima more easily
    (Notational convention)
  - parents and children: $(\mu+\lambda)$-ES → doesn't forget good solutions ("elitist selection")

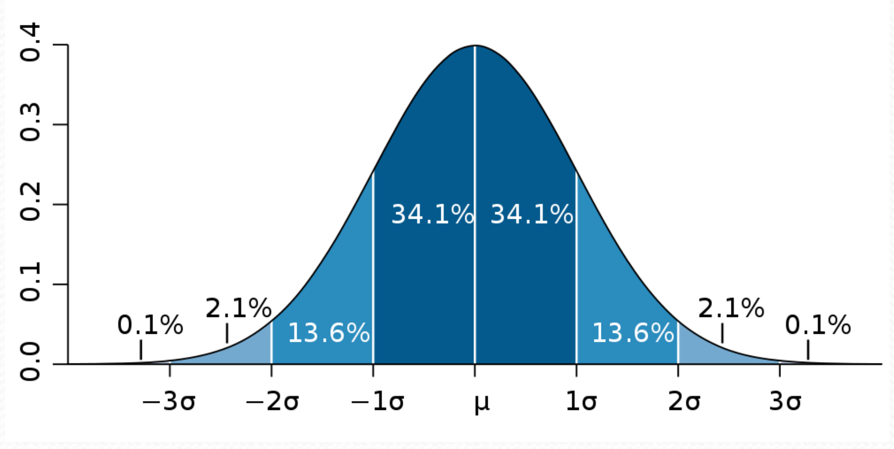
ES: Basic Mutation

- An individual is a vector $\vec{h} = (x_1, \ldots, x_n)$

- Mutate each x_i by sampling a change from a normal distribution:

$$x_i \leftarrow x_i + \Delta x_i \text{ where } \Delta x_i \underset{\uparrow}{\sim} N(0, \sigma)$$

"sampled from"



Simple modification: mutation rate for each x_i

Major question:
How to set σ or σ_i?

MAIN IDEA: make search more efficient by increasing mutation rate if this seems safe

ES: Basic Mutation

- An algorithm for setting global σ:
  - Count the number G_s of successful mutations
  - Compute the ratio of successful mutations $p_s = G_s / G$
  - Update strategy parameters according to

Improved fitness

$$\sigma_i = \begin{cases} \sigma_i / c & \text{if} \quad p_s > 0.2 \\ \sigma_i \, c & \text{if} \quad p_s < 0.2 \\ \sigma_i & \text{if} \quad p_s = 0.2 \end{cases}$$

$$c \in [0.8, 1.0]$$

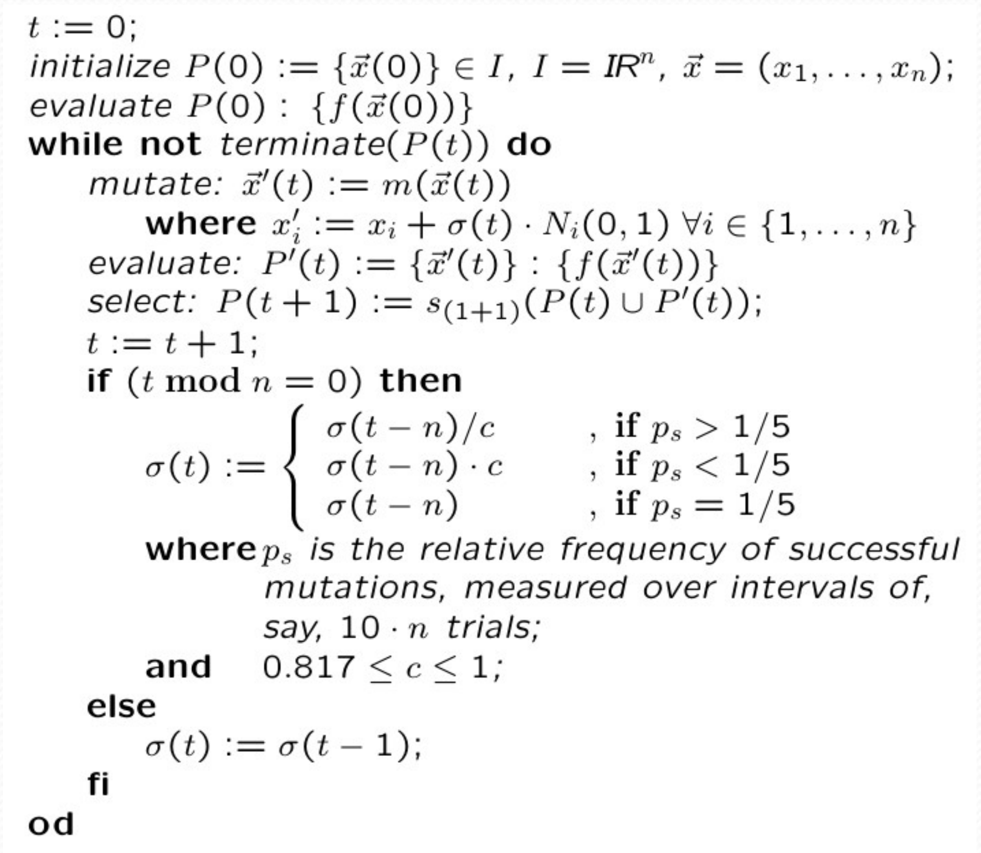
Increase mutation rate as it appears better solutions are far away

"1/5 rule"

until termination

Basic (1+1) ES

- Common use of the 1/5 rule



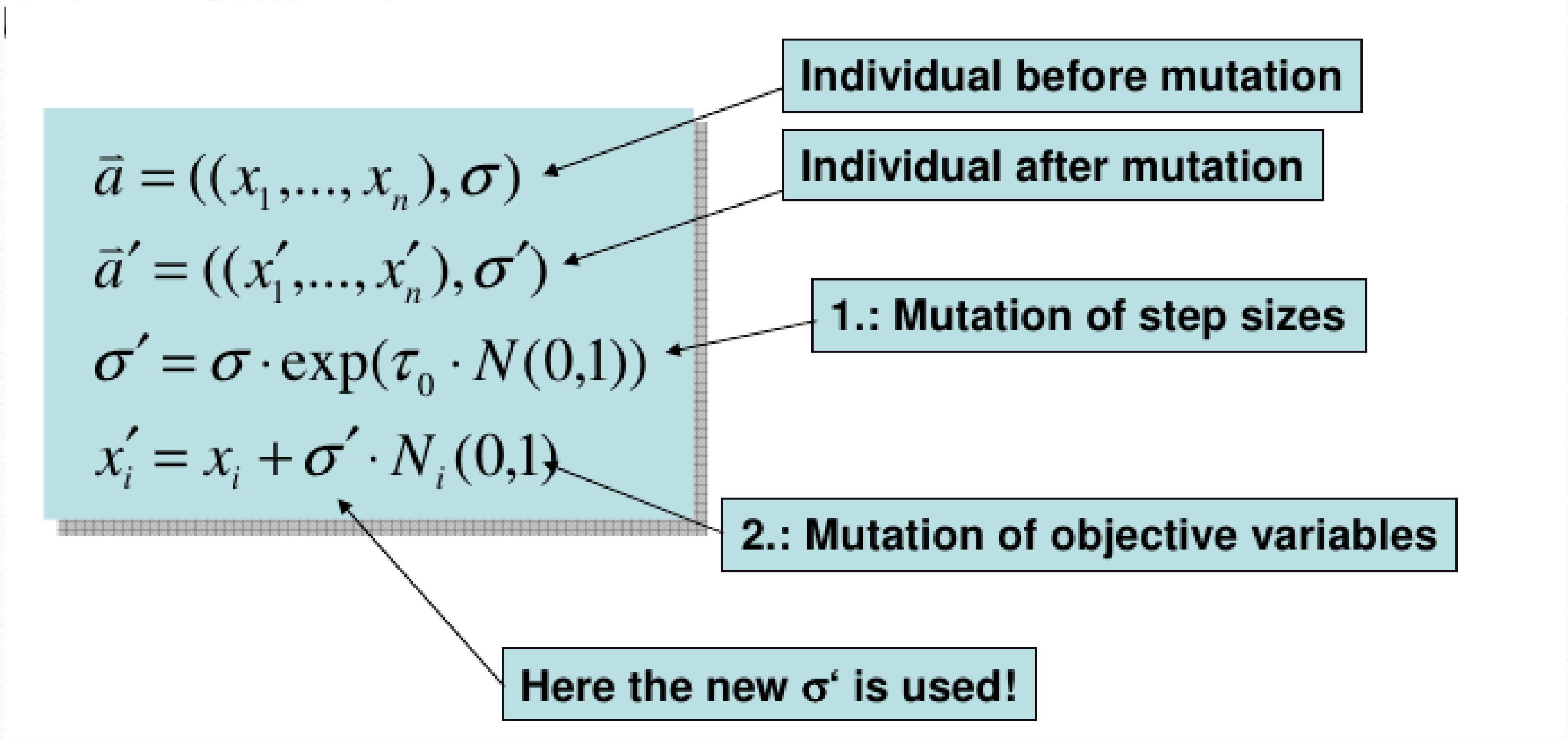
```
$t := 0$;
initialize $P(0) := \{\vec{x}(0)\} \in I,\ I = I\!R^n,\ \vec{x} = (x_1, \ldots, x_n)$;
evaluate $P(0) :\ \{f(\vec{x}(0))\}$
while not terminate$(P(t))$ do
    mutate: $\vec{x}'(t) := m(\vec{x}(t))$
        where $x_i' := x_i + \sigma(t) \cdot N_i(0,1)\ \forall i \in \{1, \ldots, n\}$
    evaluate: $P'(t) := \{\vec{x}'(t)\} :\ \{f(\vec{x}'(t))\}$
    select: $P(t+1) := s_{(1+1)}(P(t) \cup P'(t))$;
    $t := t + 1$;
    if $(t \bmod n = 0)$ then
        $\sigma(t) := \begin{cases} \sigma(t-n)/c & \text{, if } p_s > 1/5 \\ \sigma(t-n) \cdot c & \text{, if } p_s < 1/5 \\ \sigma(t-n) & \text{, if } p_s = 1/5 \end{cases}$
        where $p_s$ is the relative frequency of successful
                 mutations, measured over intervals of,
                 say, $10 \cdot n$ trials;
        and      $0.817 \leq c \leq 1$;
    else
        $\sigma(t) := \sigma(t-1)$;
    fi
od
```

ES Mutation: Strategy Parameters

- An individual is a vector $\vec{h} = (x_1, \ldots, x_n, \sigma)$ or $\vec{h} = (x_1, \ldots, x_n, \sigma_1, \ldots, \sigma_n)$
  where the σ_i are the standard deviations
- Mutate strategy parameter(s) first
  Order is important!
- If the resulting child has high fitness, it is assumed that:
  - quality of phenotype is good
  - quality of strategy parameters that led to this phenotype is good

ES Mutation: Strategy Parameters

- Mutation of one strategy parameter



$$\vec{a} = ((x_1, \dots, x_n), \sigma)$$

$$\vec{a}' = ((x_1', \dots, x_n'), \sigma')$$

$$\sigma' = \sigma \cdot \exp(\tau_0 \cdot N(0,1))$$

$$x_i' = x_i + \sigma' \cdot N_i(0,1)$$

Individual before mutation
Individual after mutation
1.: Mutation of step sizes
2.: Mutation of objective variables
Here the new σ‘ is used!

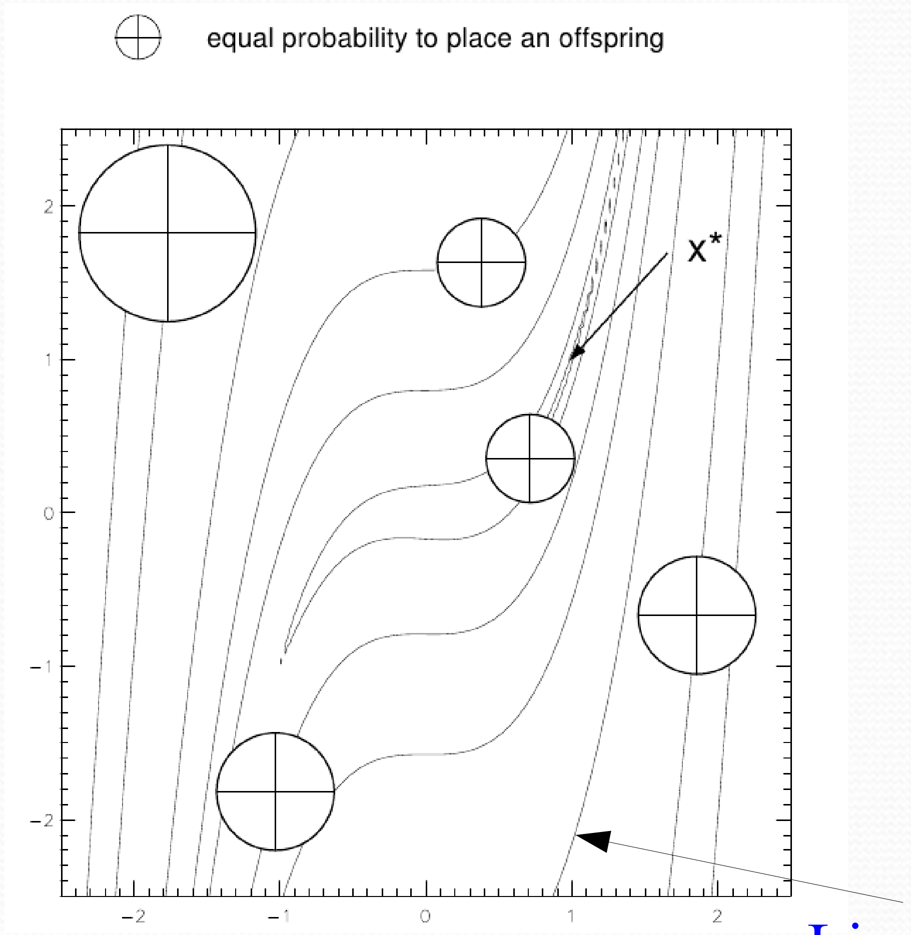
ES Mutation: Strategy Parameters

- Here τ_0 is the mutation rate
  - τ_0 bigger: faster but more imprecise
  - τ_0 smaller: slower but more imprecise
- Recommendation for setting τ_0 :

$$\tau_0 = \frac{1}{\sqrt{n}}$$

*H.-P. Schwefel: Evolution and Optimum Seeking, Wiley, NY, 1995.

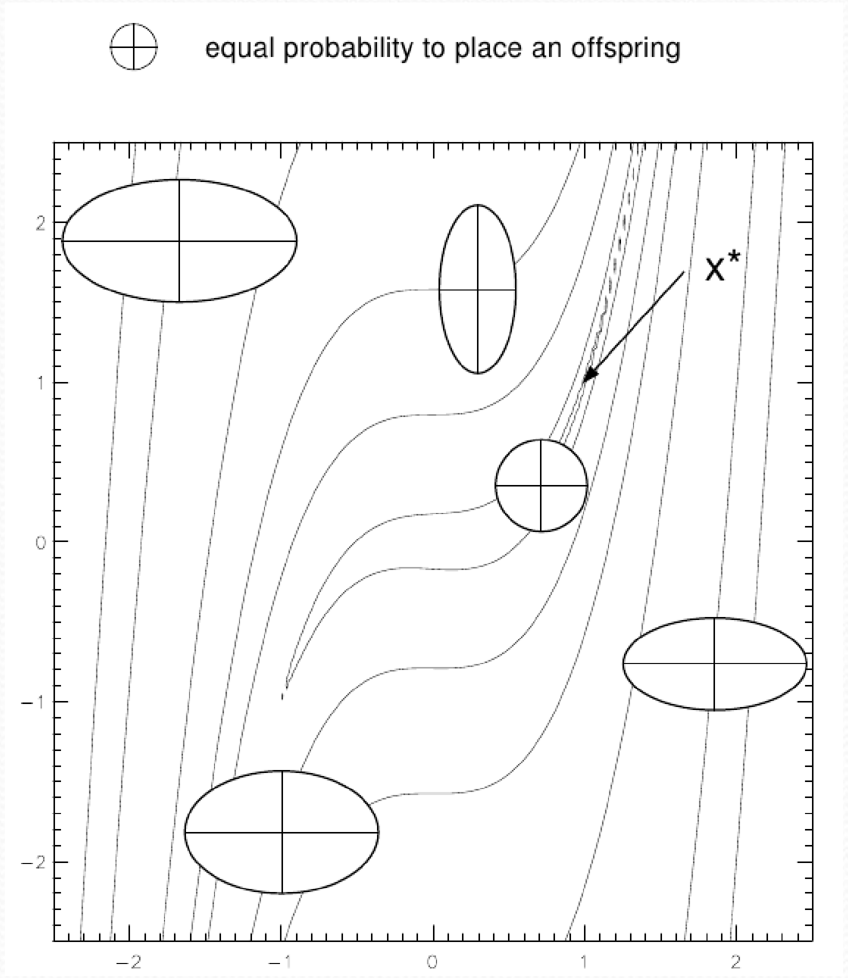
ES Mutation: Strategy Parameters



- One parameter for each individual
- 2 dimensional genotype $\vec{h} = (x_1, x_2, \sigma)$
- 5 individuals

Line indicates points with equal fitness

ES Mutation: Strategy Parameters



- One parameter for each dimension
- 2 dimensional genotype

$$\vec{h} = (x_1, x_2, \sigma_1, \sigma_2)$$

- 5 individuals

ES Mutation: Strategy Parameters

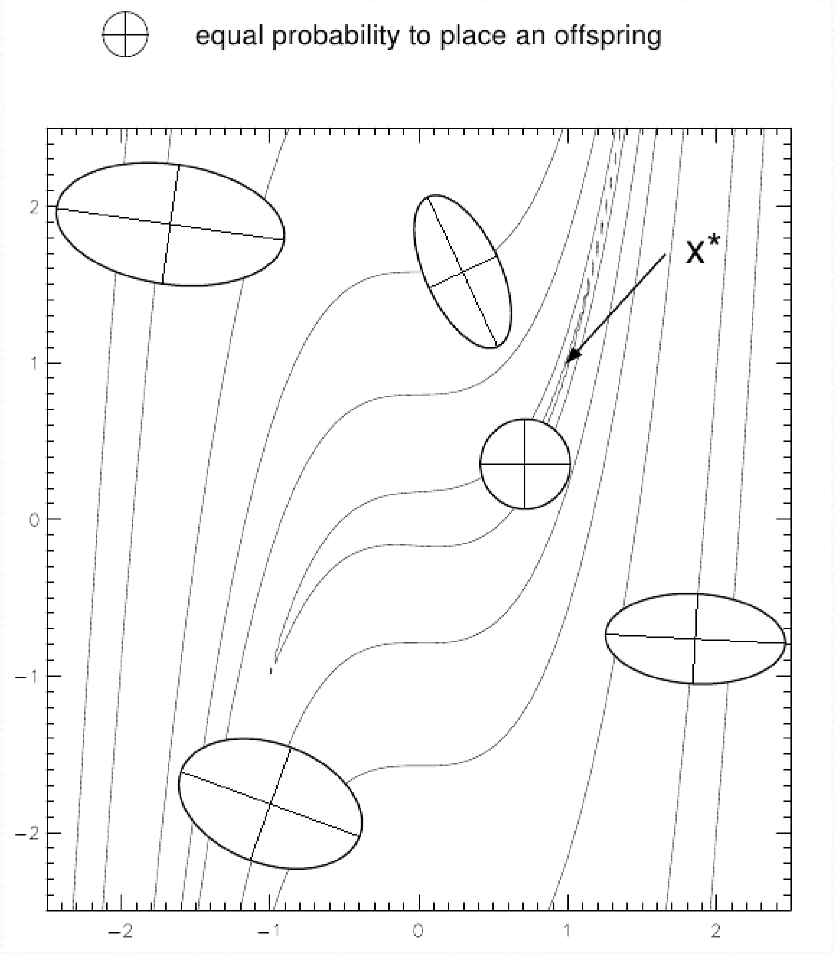
- Mutation of all strategy parameters

$$\begin{aligned}\sigma_i' &= \sigma_i \cdot \exp(\tau' \cdot N(0,1) + \tau \cdot N_i(0,1)) \\ x_i' &= x_i + \sigma_i' \cdot N_i(0,1)\end{aligned}$$

Sample from normal distribution,
the same for all parameters

Update for this specific parameter

ES Mutation: Strategy Parameters



- An individual is a vector

$$\vec{h} = (x_1, \ldots, x_n, \sigma_1, \ldots, \sigma_n, \alpha_1, \ldots, \alpha_m)$$

where α_i encode angles

- Also here mutation can be defined

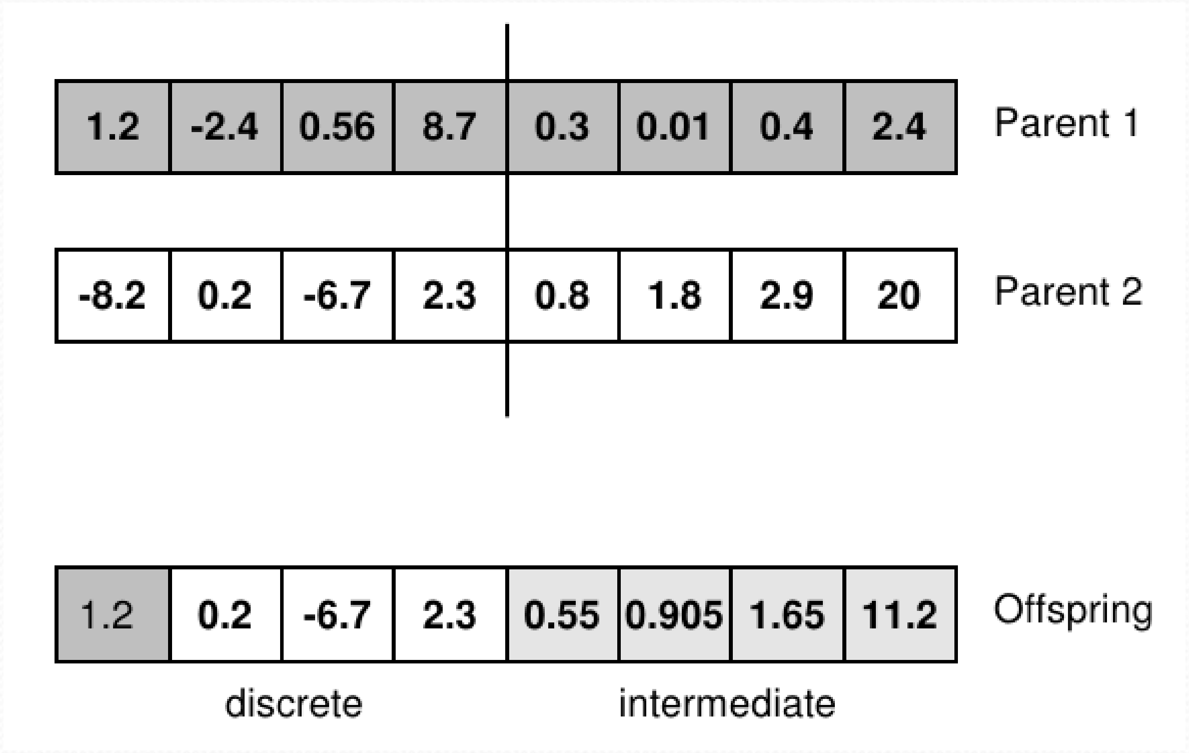
- Mathematical details skipped

ES Crossover / Recombination

- Application of operator creates **one** child (not two)
- Is applied λ times to create an offspring population of λ size (on which then mutation and selection is applied)
- Per offspring gene two parent genes are involved
- Choices:
  - combination of two parent genes:
    - average value of parents (*intermediate recombination*)
    - value of one randomly selected parent (*discrete recombination*)
  - choice of parents:
    - a different pair of parents for each gene (*global recombination*)
    - the same pair of parents for all genes

ES Crossover / Recombination

- Default choice: discrete recombination on phenotype, intermediate recombination on strategy parameters



1.2	-2.4	0.56	8.7	0.3	0.01	0.4	2.4	Parent 1
-8.2	0.2	-6.7	2.3	0.8	1.8	2.9	20	Parent 2
1.2	0.2	-6.7	2.3	0.55	0.905	1.65	11.2	Offspring

GAs vs. ES

Genetic algorithms

- Crossover is the main operator
- Uses also mutation
- Encoding for problem representation
- Biased selection of the parents
- Algorithm parameters often fixed
- Selection → Crossover → Mutation

Evolution strategies

- Mutation is the main operator
- Uses also crossover (recombination)
- No encoding needed for problem representation
- Random selection of the parents
- Adaptive set of algorithm parameters (strategy parameters)
- Crossover → Mutation → Selection

Genetic Programming

- Goal: to learn computer programs from examples (like in machine learning and data mining)
- Main idea:
  represent (simple) *computer programs* in individuals of arbitrary size
- Redefinitions of
  - selection
  - crossover
  - mutation

Individuals are Program Trees / Parse Trees

- Representation of
  - Arithmetic formulas

$$2 \cdot \pi + \left((x+3) - \frac{y}{5+1} \right)$$

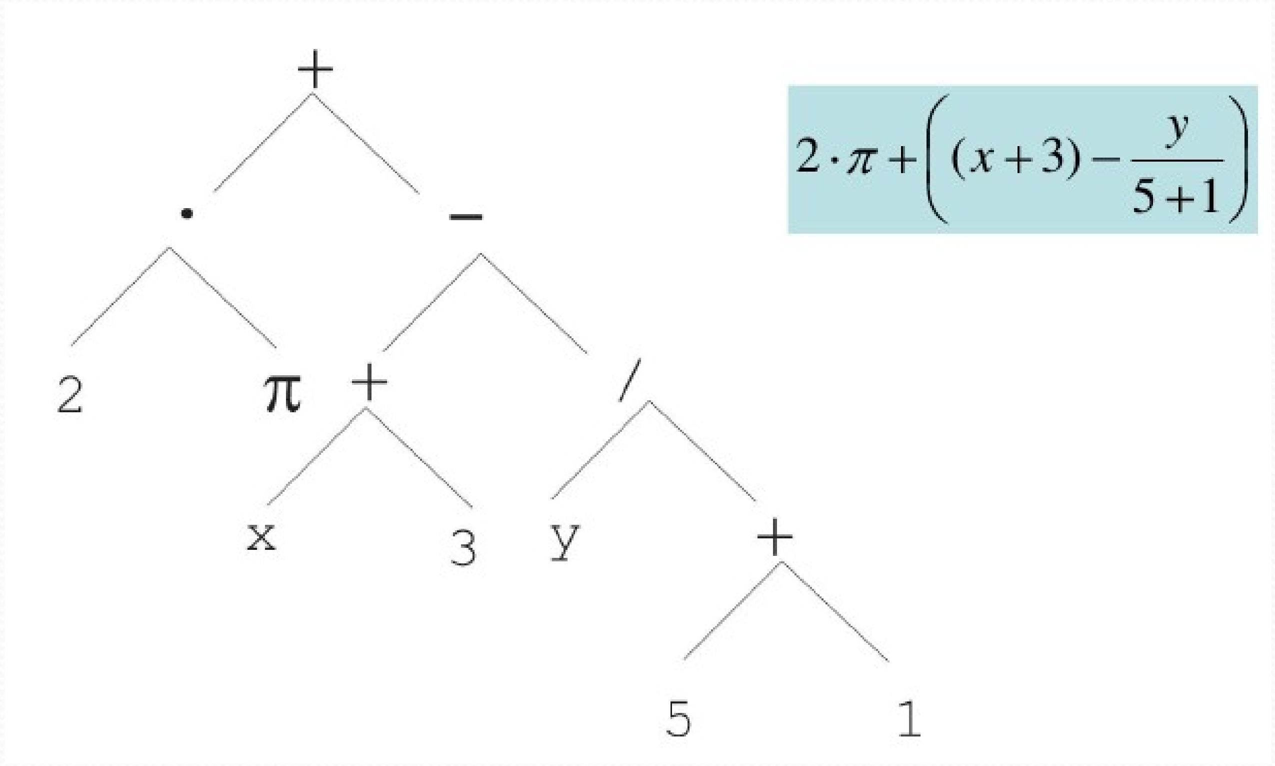
  - Logical formulas

$$(x \wedge \text{true}) \rightarrow ((x \vee y) \vee (z \leftrightarrow (x \wedge y)))$$

  - Computer programs

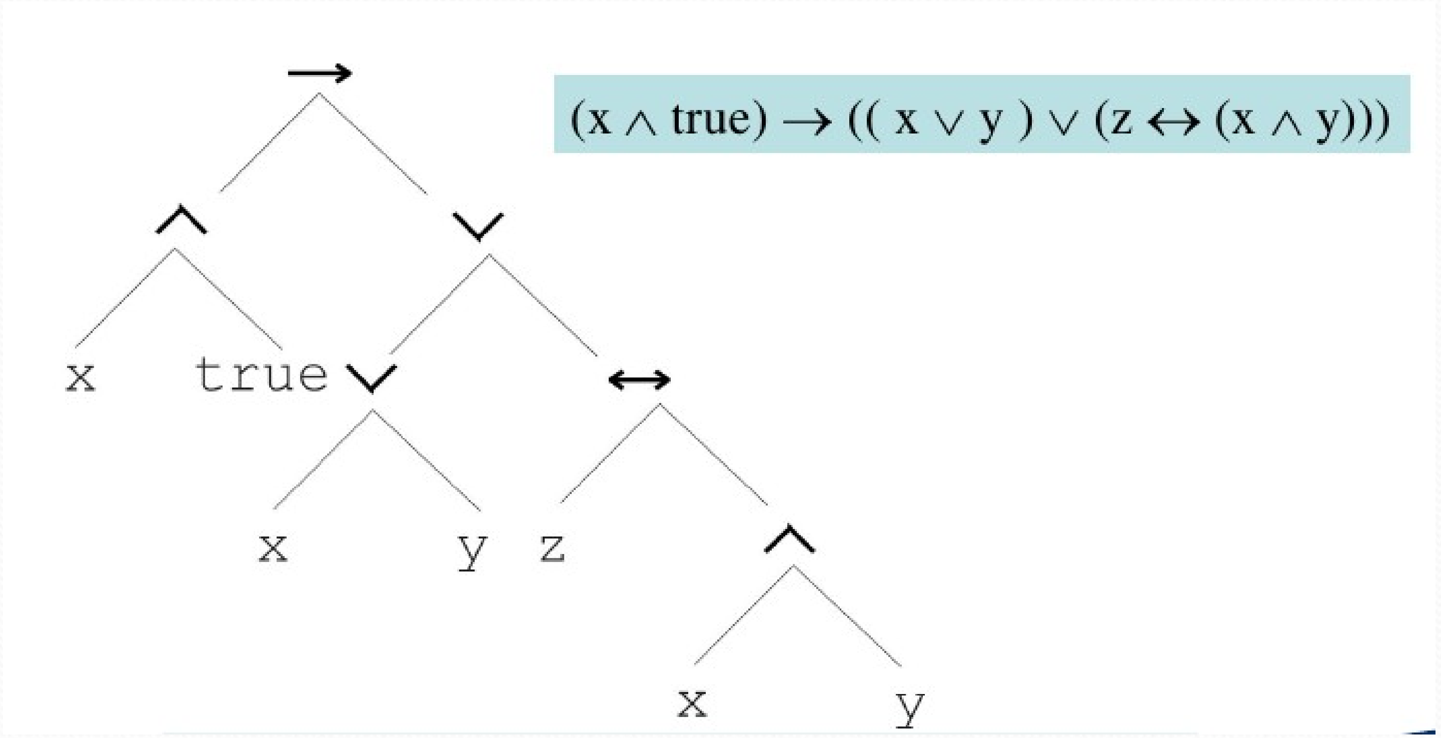
```
i =1;
while (i < 20)
{
          i = i +1
}
```

Representation of Arithmetic Formula as Tree



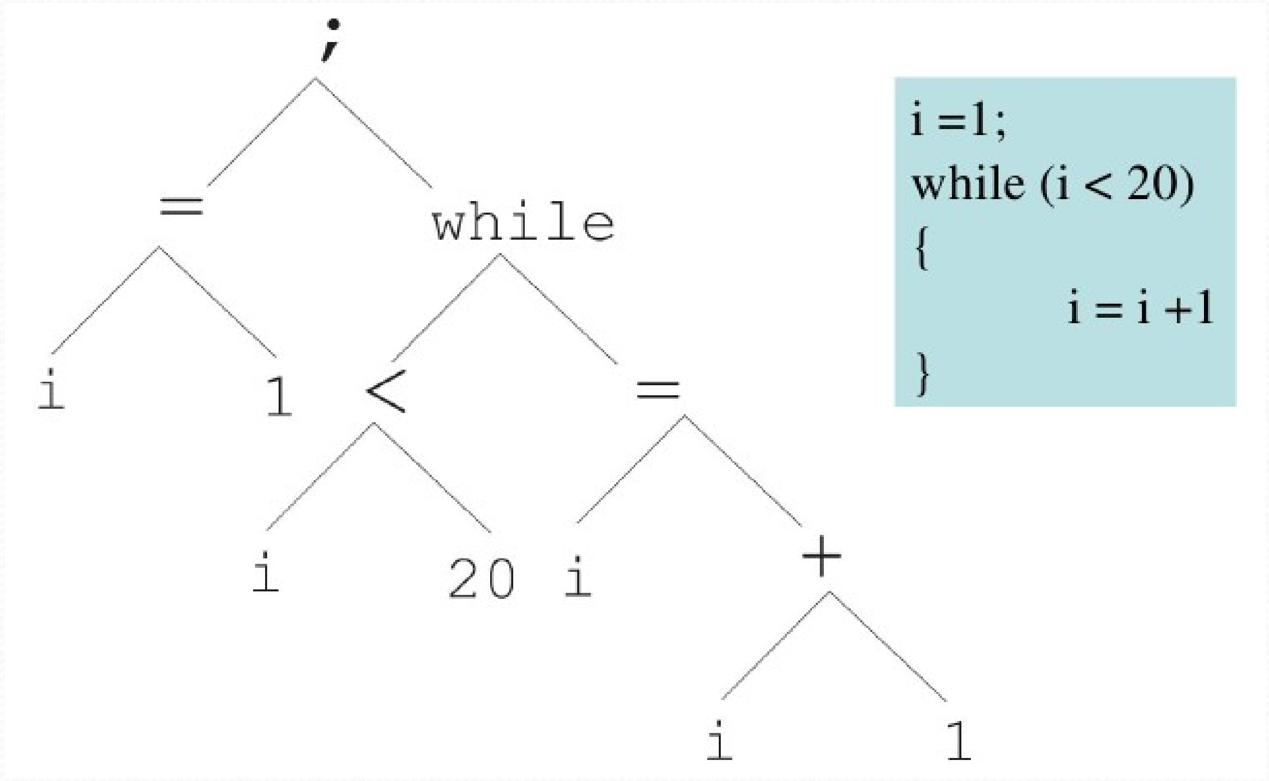
$$2 \cdot \pi + \left((x+3) - \frac{y}{5+1} \right)$$

Representation of Logical Formula



$(x \wedge \text{true}) \rightarrow ((x \vee y) \vee (z \leftrightarrow (x \wedge y)))$

Representation of Computer Programs



```
i =1;
while (i < 20)
{
        i = i +1
}
```

Representation

- Trees consisting of:
  - terminals (leaves)
    - constants
    - variables (inputs to the program/formula)
  - functions of fixed arity (internal nodes)

Considerations in Function Selection

- **Closure:** any function should be well-defined for all arguments

  Example: { *, / } is not closed as division is not well defined if the second argument is 0 → redefine /.

- **Sufficiency:** the function and terminal set should be able to represent a desirable solution

Evolutionary Cycle

- Fixed population size
- Create a new population by randomly selecting an operation to apply, each of which adds one or two individuals into the new population, starting from one or two fitness proportionally selected individuals:
  - reproduction (copying)
  - one of many crossover operations
  - one of many mutation operations

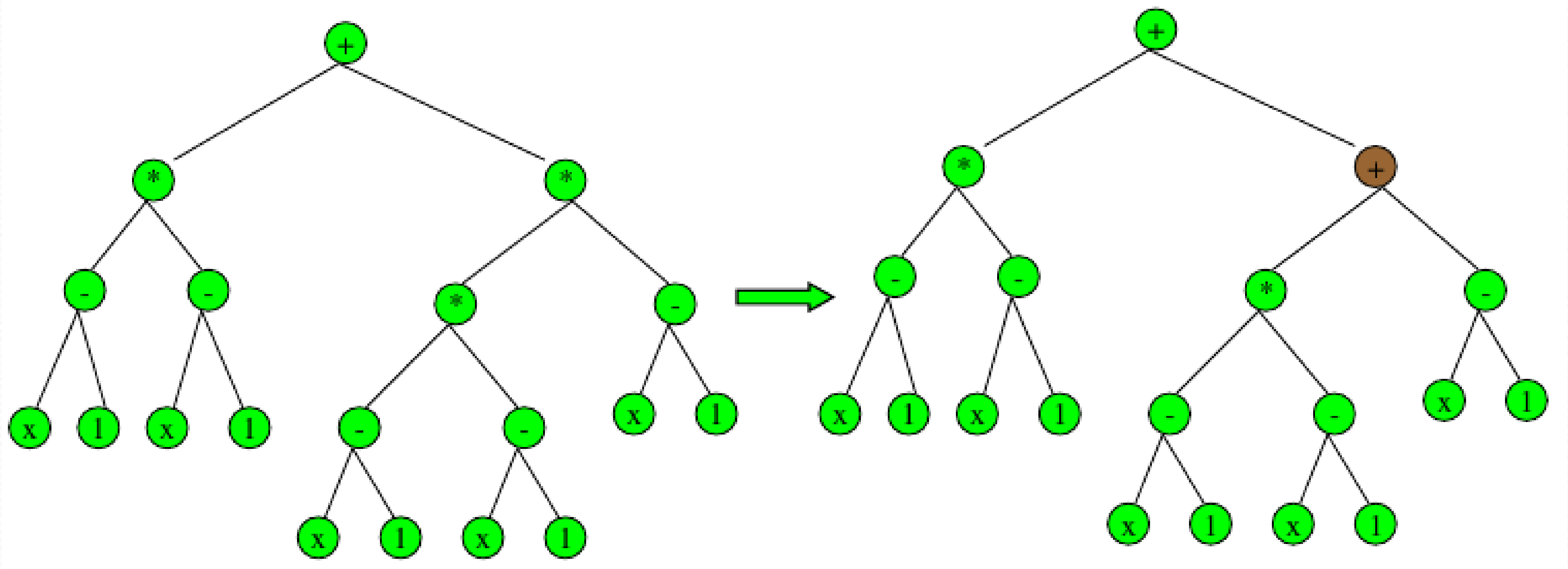
Initialization

- Given is a maximum depth on trees D_{max}
- Full method:
    - for each level < D_{max} insert a node with function symbol (recursively add children of appropriate types)
    - for level D_{max} insert a node with a terminal
- Grow method:
    - for each level < D_{max} insert a node with either a terminal or a function symbol (and recursively add children of appropriate types to these nodes)
    - for level D_{max} insert a node with a terminal
- Combined method: half of the population full, the other grown

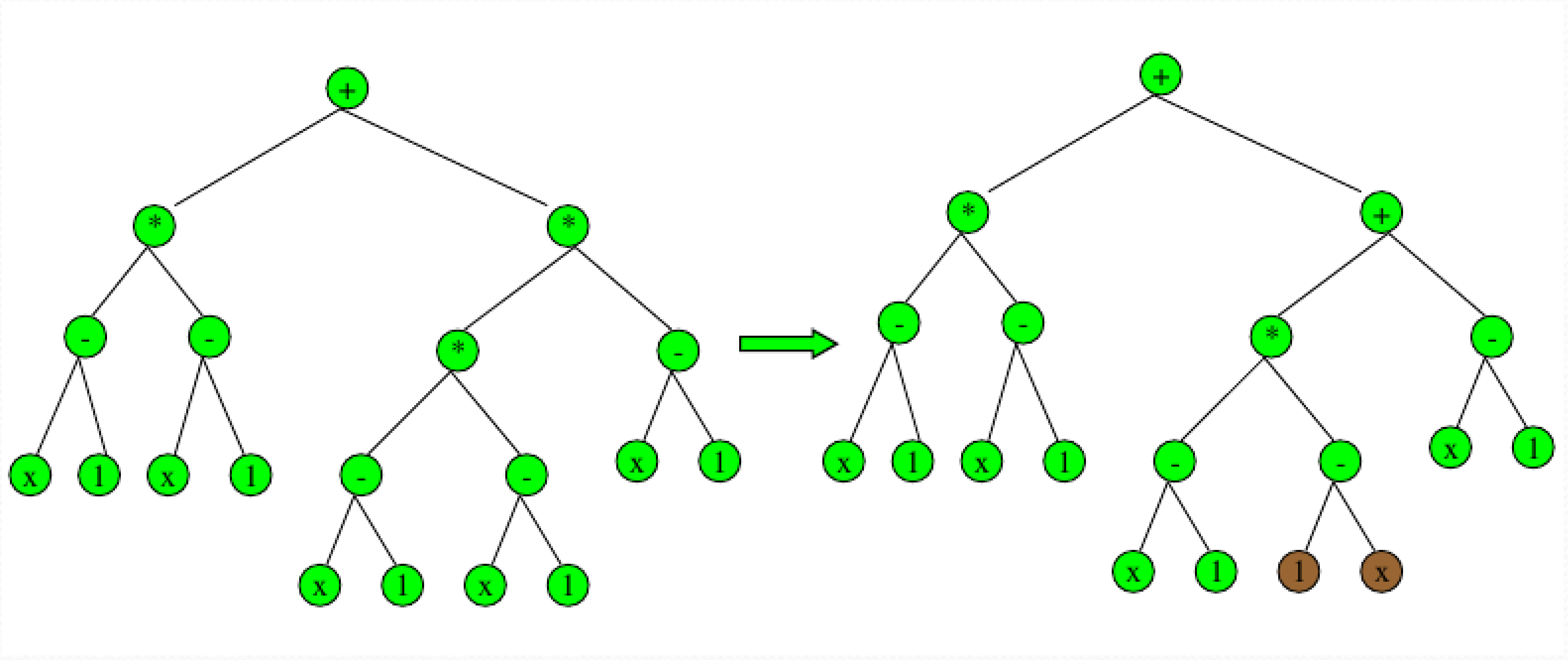
Mutation

Operator name	Description
Point mutation	single node exchanged against random node of same class
Permutation	arguments of a node permuted
Hoist	new individual generated from subtree
Expansion	terminal exchanged against random subtree
Collapse subtree	subtree exchanged against random terminal
Subtree mutation	subtree exchanged against random subtree

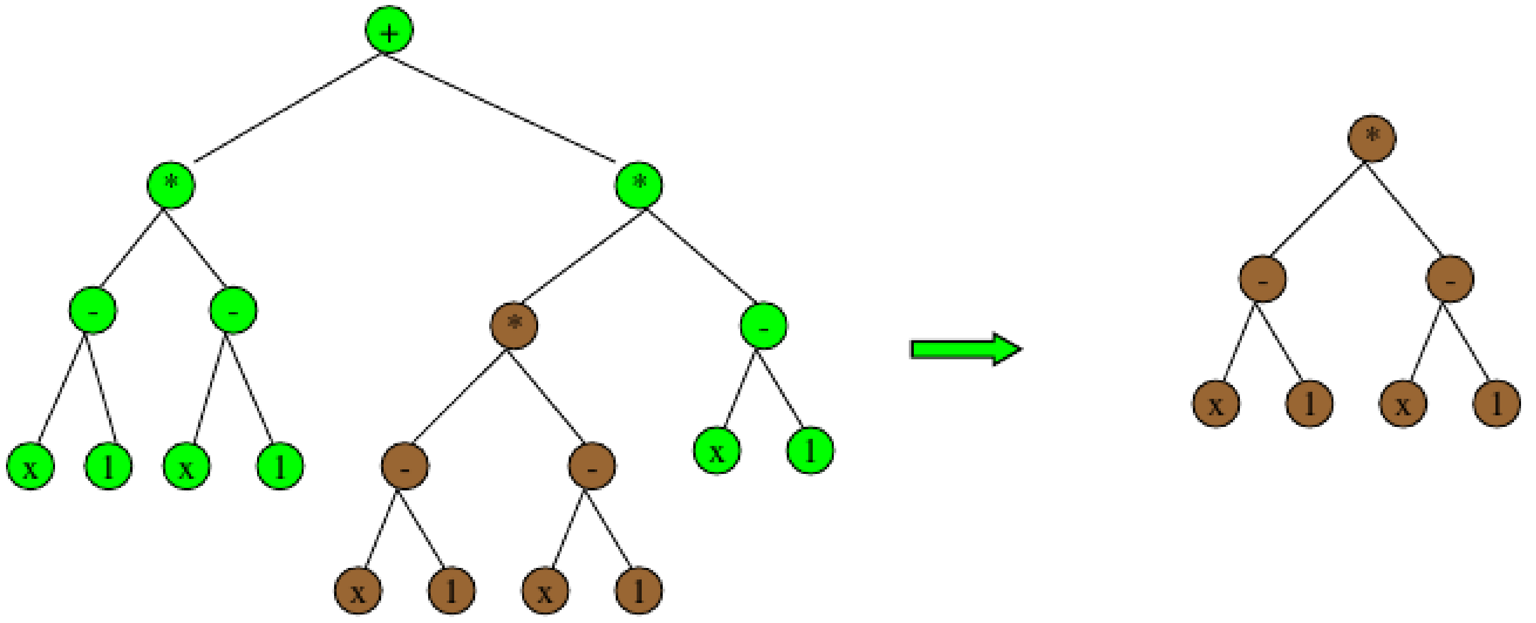
Point Mutation



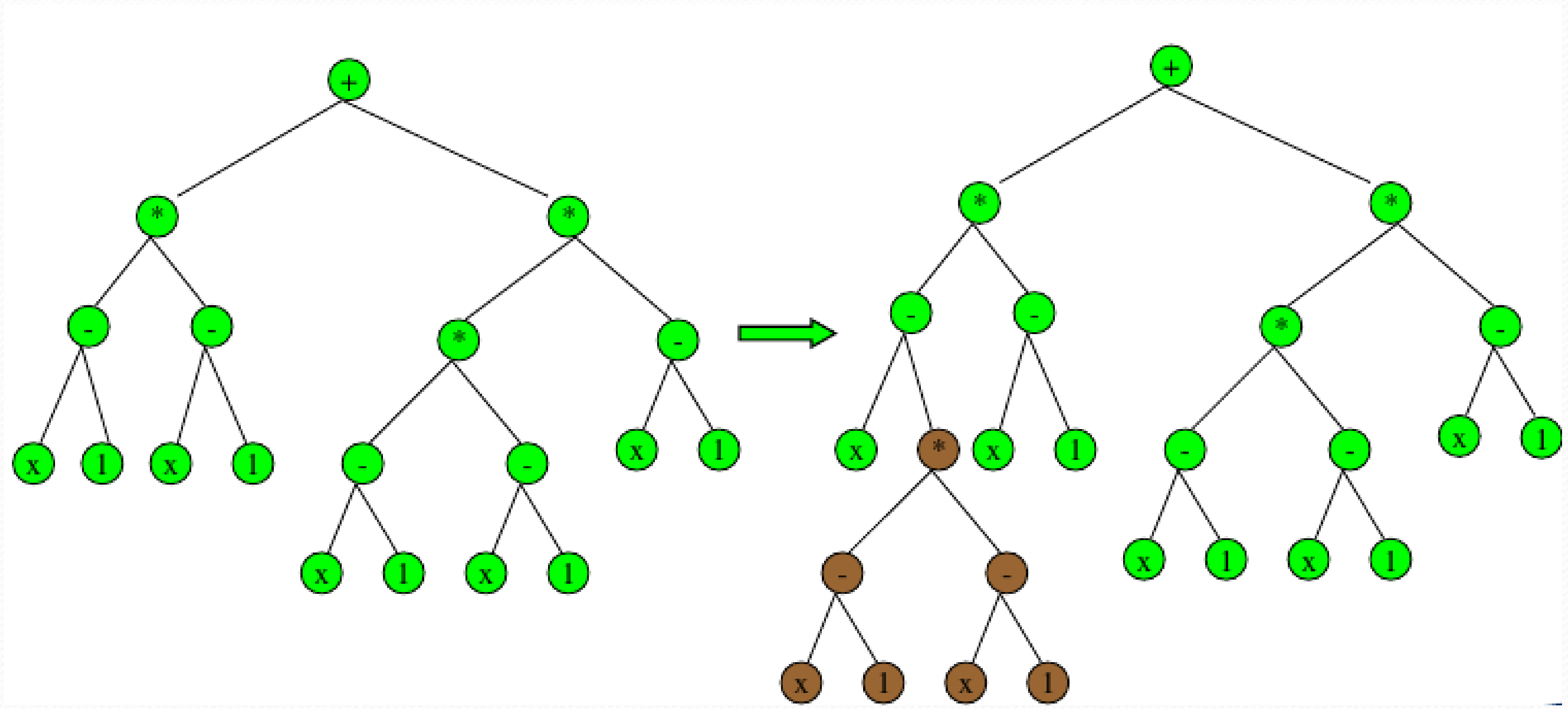
Permutation



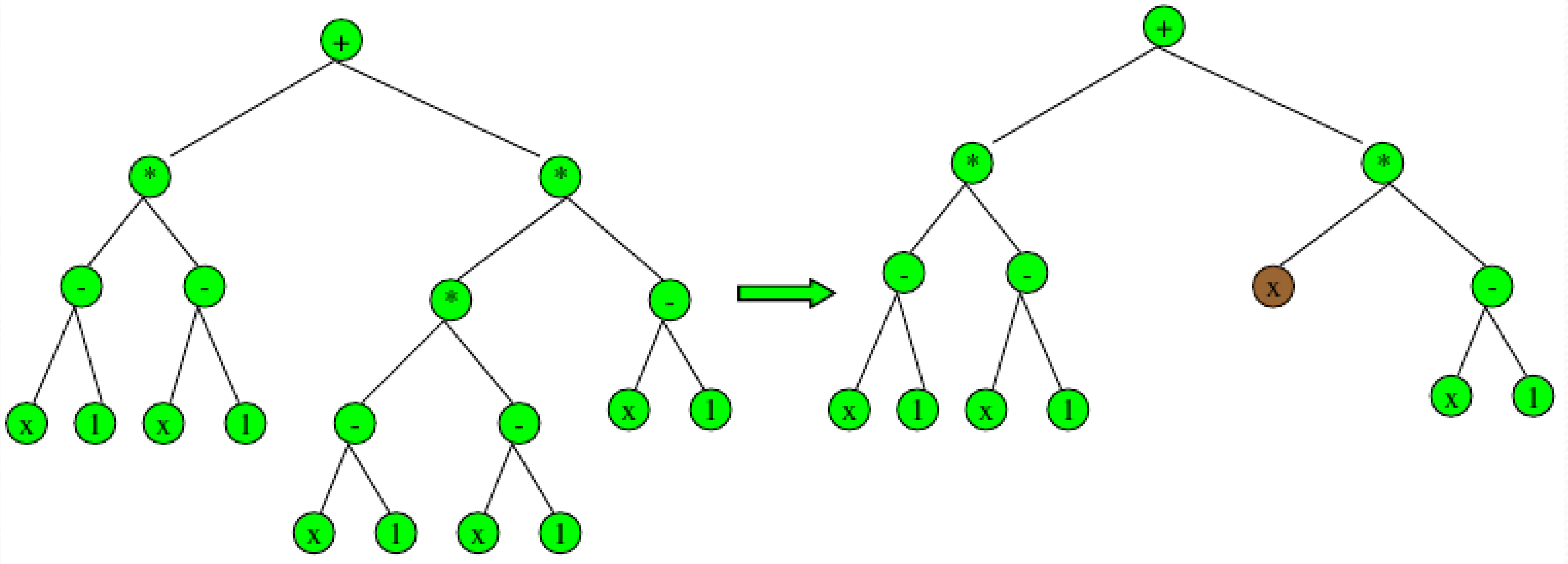
Hoist



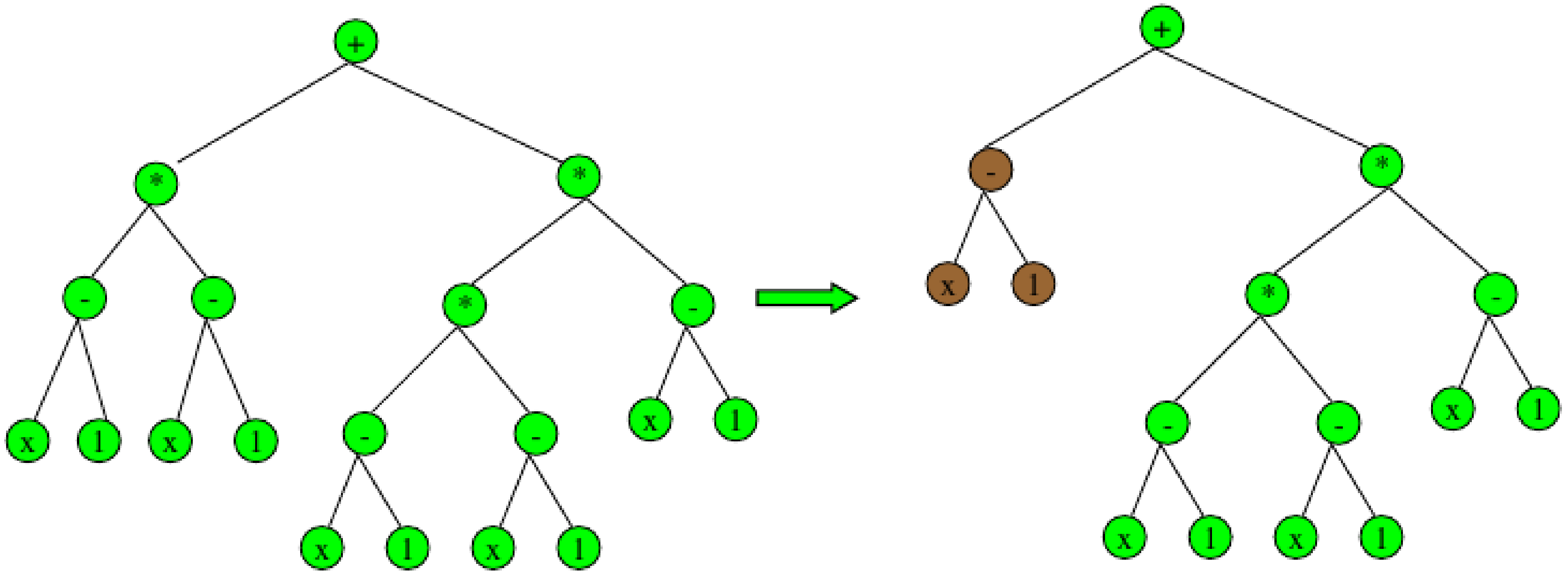
Expansion Mutation



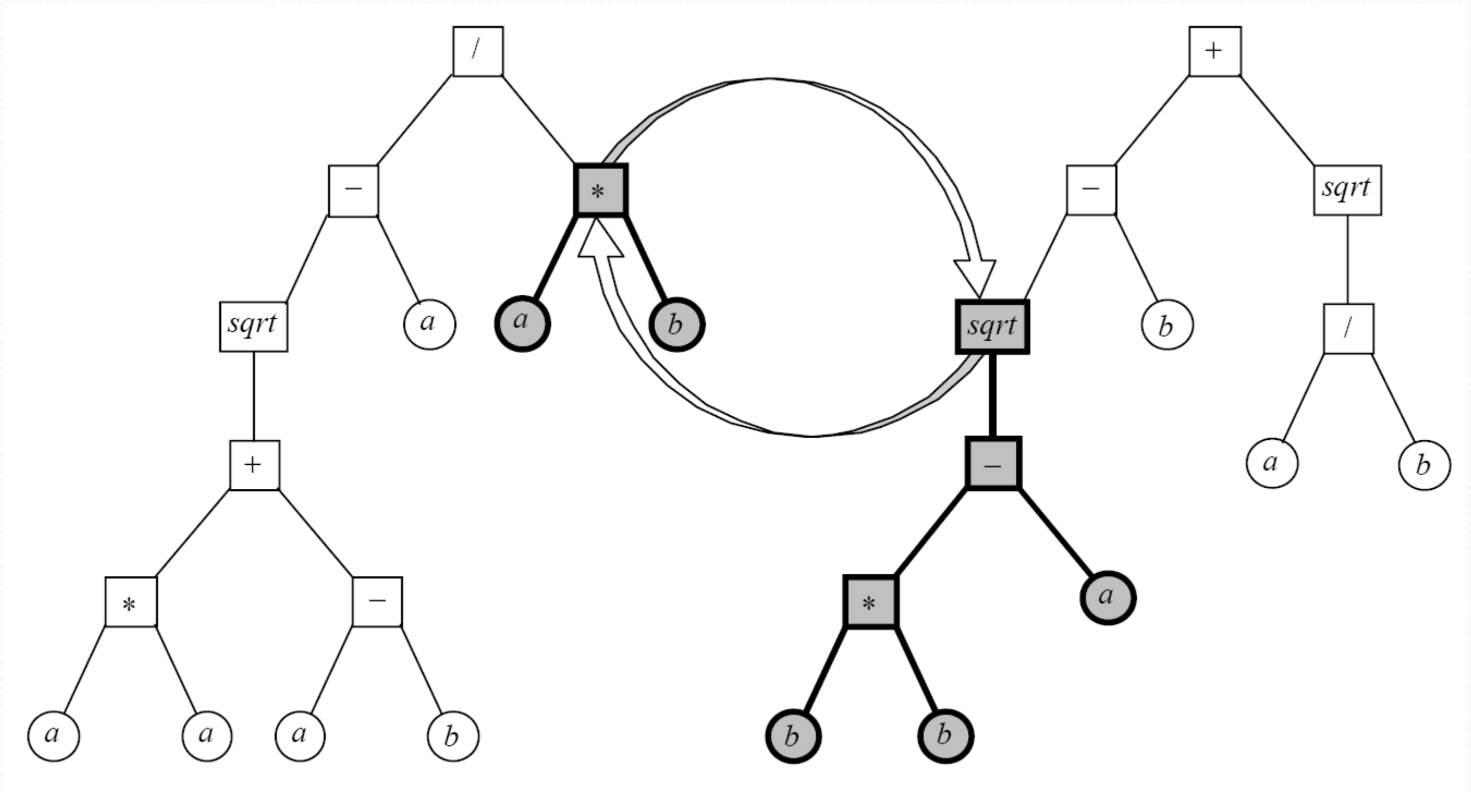
Collapse Subtree Mutation



Subtree Mutation



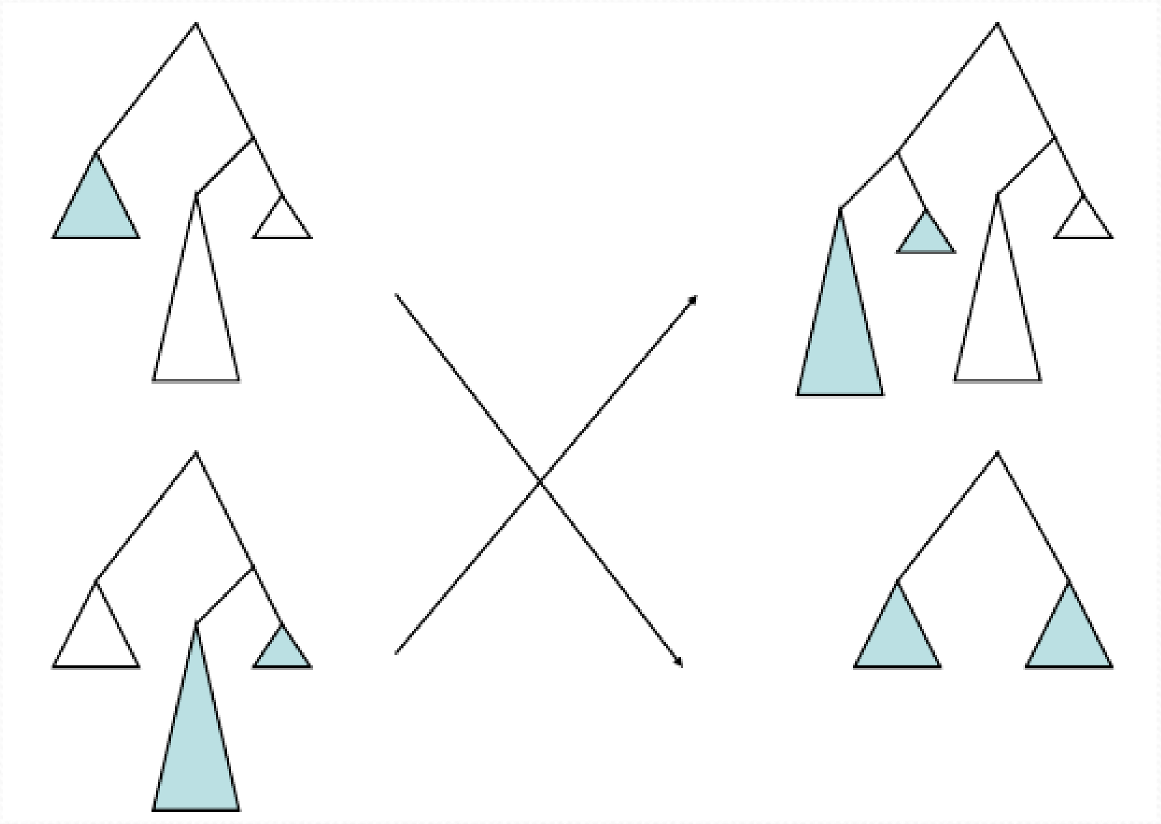
Crossover



(/ (− (*sqrt* (+ (* *a a*) (− *a b*))) *a*) (* *a b*))

(+ (− (*sqrt* (− (* *b b*) *a*)) *b*) (*sqrt* (/ *a b*)))

Self-Crossover



Bloat

- “Survival of the fattest”, i.e. the tree sizes in the populations increase over time

- Countermeasures:
  - simplification
  - penalty for large trees
  - hard constraints on the size of trees resulting from operations

Editing Operator

- An operation that simplifies expressions
- Examples:
  - X AND X → X
  - X OR X → X
  - NOT(NOT(X)) → X
  - X + 0 → X
  - X . 1 → X
  - X . 0 → 0
  - ….

Example – Symbolic Regression Pythagorean Theorem

Not (necessarily) linear

Negnevitsky 2004

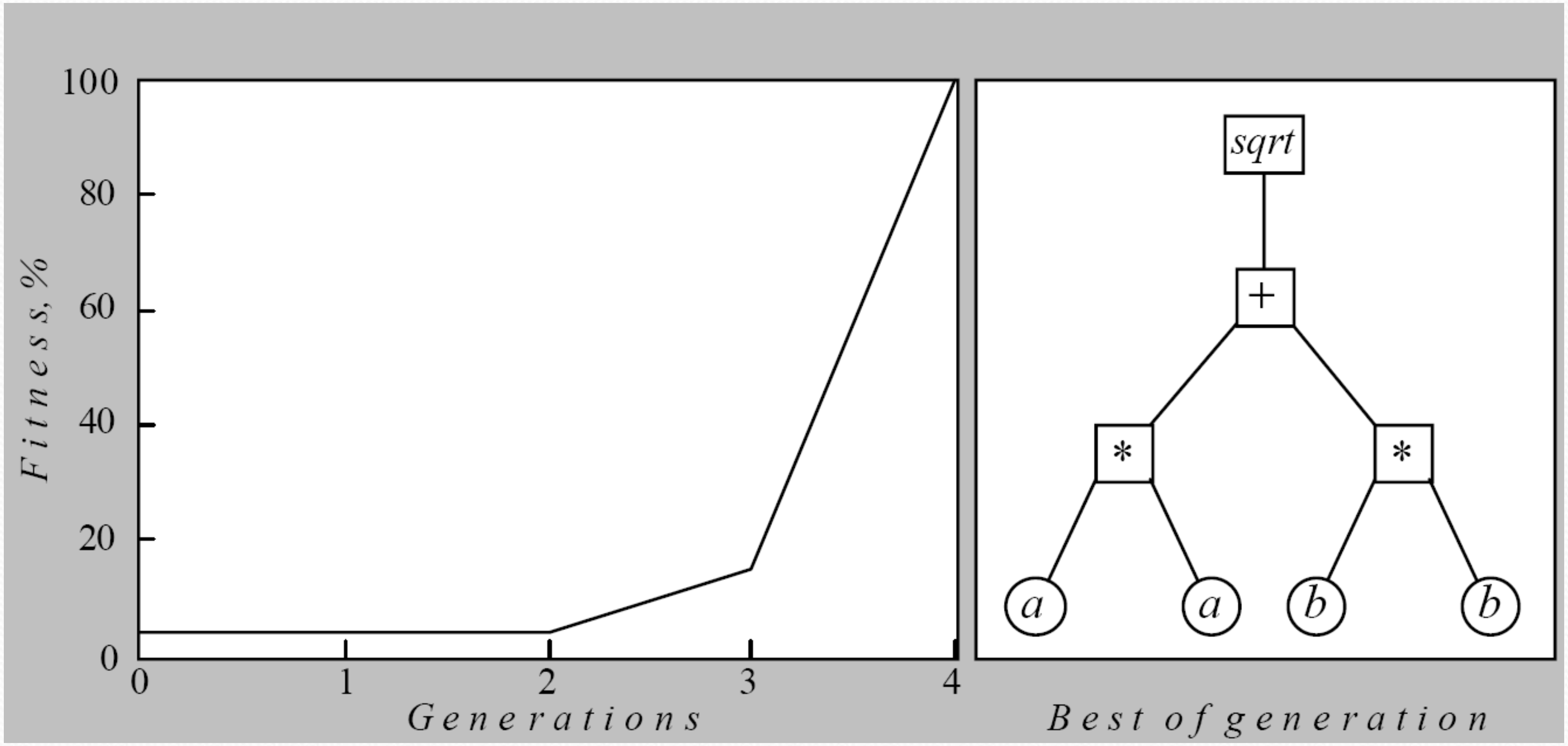
Underlying function: $c = \sqrt{a^2 + b^2}$

Fitness cases:

Side a	Side b	Hypotenuse c	Side a	Side b	Hypotenuse c
3	5	5.830952	12	10	15.620499
8	14	16.124515	21	6	21.840330
18	2	18.110770	7	4	8.062258
32	11	33.837849	16	24	28.844410
4	3	5.000000	2	9	9.219545

Language elements: +, -, *, /, sqrt, *a*, *b*

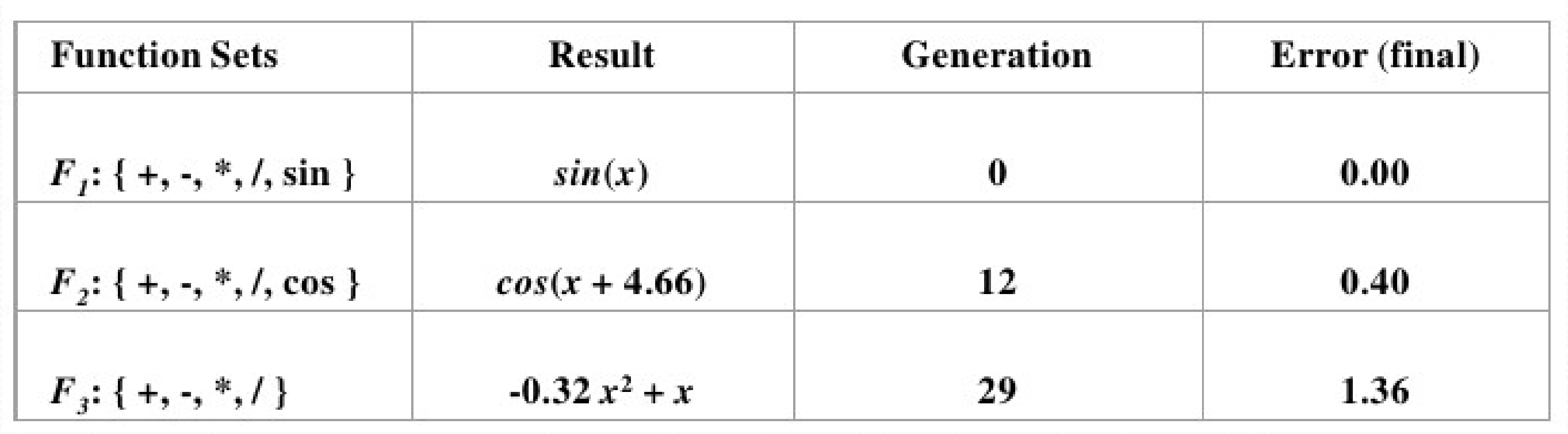
Results



100
80
60
40
20
0
Fitness, %
0
1
2
3
4
Generations
sqrt
+
*
*
a
a
b
b
Best of generation

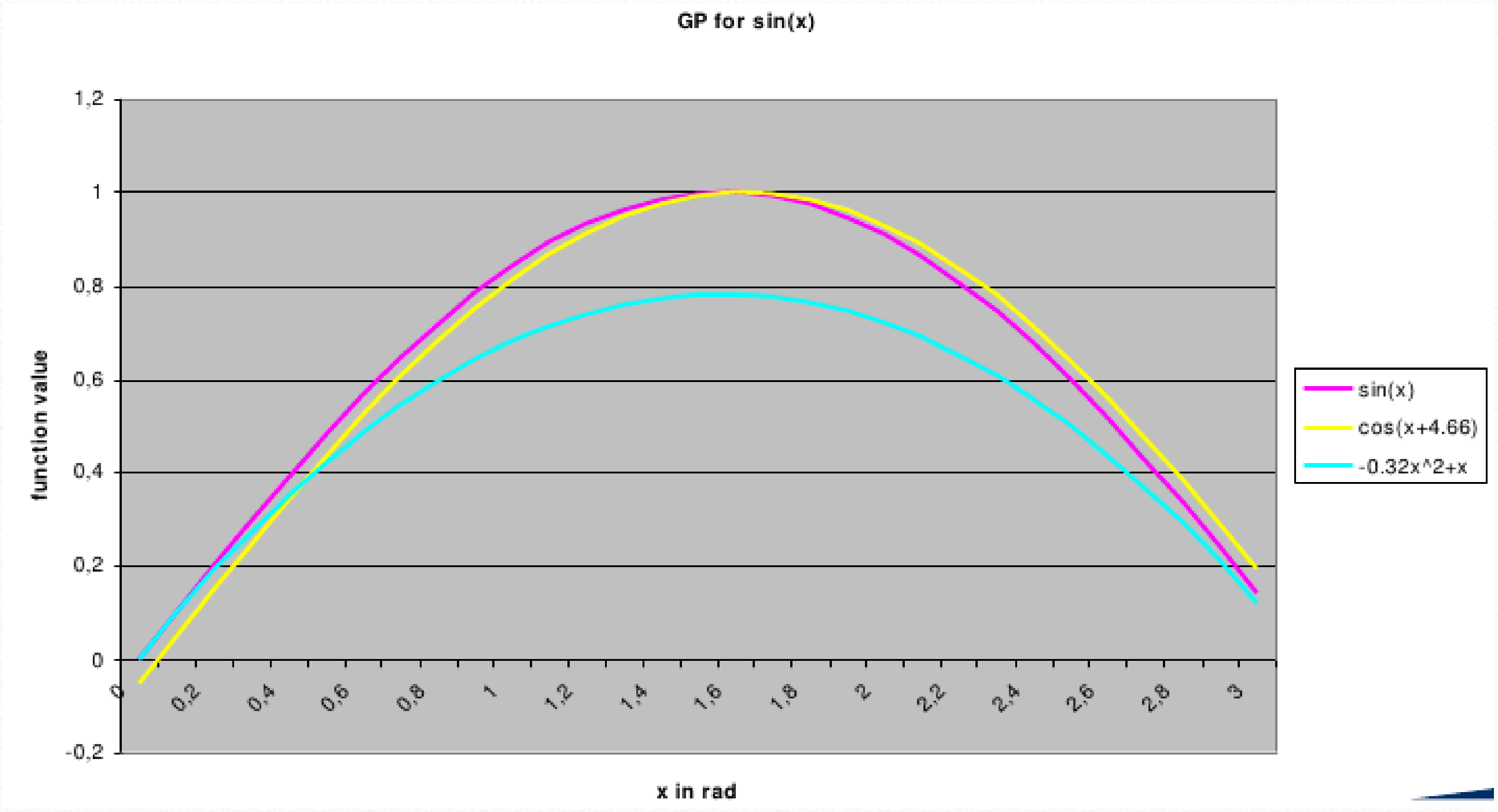
Example – Symbolic Regression Approximation of sin(x)

- **Given** examples (x,sin(x)) with x in {0,1,…,9}
- **Find** a good approximation of sin(x)



Function Sets	Result	Generation	Error (final)
F_1: { +, -, *, /, sin }	$sin(x)$	0	0.00
F_2: { +, -, *, /, cos }	$cos(x + 4.66)$	12	0.40
F_3: { +, -, *, / }	$-0.32\,x^2 + x$	29	1.36

Example – Symbolic Regression Approximation of sin(x)



GAs vs. GP

Genetic algorithms	Genetic programming
Chromosomes represent coded solutions	Chromosomes represent executable code
Fixed length chromosomes	Variable length chromosomes
A small set of well-defined genetic operators	More complex genetic operators required
Conceptually simple	Conceptually complex
Fixed order of operators	Order of operators not fixed