

Intelligent Systems Engineering

EC5 Evolutionary Algorithms

Genetic Algorithms

Petr Musilek

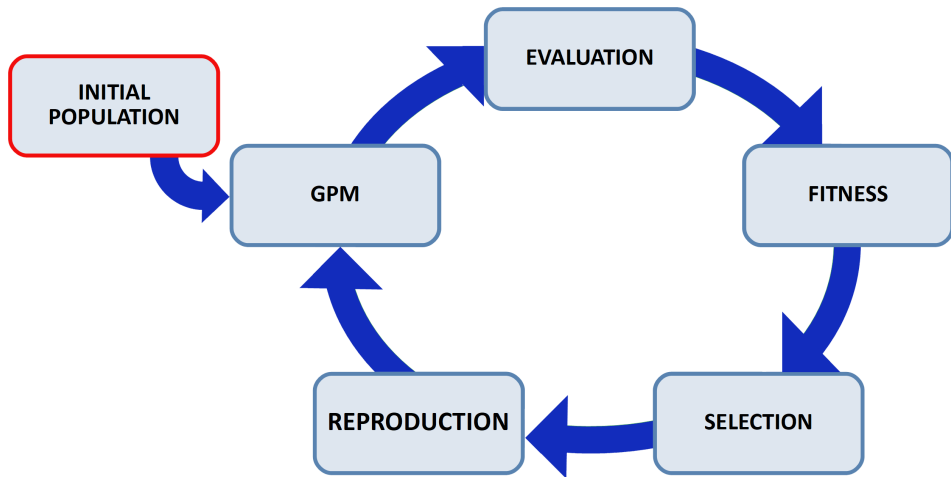
University of Alberta

Outline

- 1 Review of Evolutionary Algorithms
- 2 Genetic Algorithms
 - Canonical Genetic Algorithm
 - Classification
- 3 Fixed-Length String Chromosomes
 - Creation
 - Mutation
 - Crossover
- 4 Variable-Length String Chromosomes
- 5 Schema Theorem

These notes are based on [Weise 2011] chapter 29

Execution cycle of GA



Generic Evolutionary Algorithm

Let $t = 0$ be the generation counter;

Create and initialize an n_x -dimensional population, $\mathcal{C}(0)$, to consist of n_s individuals;

while *stopping condition(s) not true* **do** 

 Evaluate the fitness, $f(x_i(t))$, of each individual, $x_i(t)$;

 Perform reproduction to create offspring;

 Select the new population, $\mathcal{C}(t + 1)$;

 Advance to the new generation, i.e. $t = t + 1$;

end while

Genetic Algorithms

Genetic algorithms apply the ideas of genetic evolution to solving optimization problems:

individuals \equiv candidate solutions

- Individuals of a species exist within a population (not alone)
- Populations evolve through generations:
there is a parent generation, followed by an offspring generation
- The fitness of each individual determines its probability of survival during selection
- Individuals that survive become the parents of the next generation

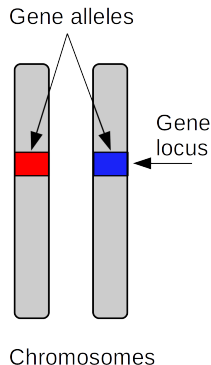
GAs: Terminology

The search spaces \mathbb{G} of Genetic Algorithms are referred to as **genomes** that are strings or linear sequences.

- **Genotypes**, the elements of genomes, are also called **chromosomes** because of their linear structure.

Terminology

- **Gene** - a basic informational unit in a genotype g (a sub-string in the chromosome).
- **Alleles** - the values, or the states, that genes might have. (e.g. 0 or 1 in the case of binary encoding)
- **Locus** - the position where a specific gene can be found in a chromosome.



Canonical Genetic Algorithm

The canonical GA (CGA) as proposed by [Holland 1975] follows the general algorithm (slide 4), with the following implementation specifics:

- A bitstring representation was used. *Binary*
- Proportional selection was used to select parents for recombination.
- One-point crossover was used as the primary method to produce offspring.
- Uniform mutation was proposed as a background operator of little importance.



less important

Since the CGA, several variations have been developed that differ in

- representation scheme,
- selection operator,
- crossover operator, and
- mutation operator.

and some other nature-inspired concepts such as mass extinction, population islands, etc.

Classification according to Chromosome Length

Fixed-length tuple

- The loci i of genes $g[i]$ are *constant*. Each gene can have a fixed meaning and tuples may contain elements of different types \mathbb{G}_i .

$$\mathbb{G} = \{\forall (g[0], g[1], \dots, g[n-1]) : g[i] \in \mathbb{G}_i \ \forall i \in 0..n-1\}$$

$$\mathbb{G} = \mathbb{G}_0 \times \mathbb{G}_1 \times \dots \times \mathbb{G}_{n-1}$$

Variable-length list


- In *variable* length string genomes the positions of the genes shift after applying reproduction operations. All elements have the same type \mathbb{G}_T .

$$\mathbb{G} = \{\forall \text{ lists } g : g[i] \in \mathbb{G}_T \forall 0 \leq i < \text{len}(g)\}; \ \mathbb{G} = \mathbb{G}_T^*$$

Classification according to encoding type

- Parameters of the solutions (genes) are concatenated to form a string (chromosomes)
- Encoding is a data structure representing the candidate solutions
- Good encoding is very important for the performance of a GA

There are three types of encoding

- Binary encoding
- Value encoding 
- Permutation encoding

Examples:

- *Binary-coded* algorithms are GAs with bit string genomes (incl. *gray-coded* algorithms)
- *Integer-coded* algorithms are GAs with integer numeric vector genomes.
- *Real-coded* algorithms are GAs with real numeric vector genomes.
- *Permutation-based* algorithms for combinatorial optimization problems

Creation: Fully-Random Genotypes

Creation is a nullary reproduction operation.

To create a *random, fixed-length string* means to initialize a tuple with a structure defined by a genome with random values.

Algorithm 29.1: $g \leftarrow \text{createGAFLBin}(n)$

Input: n : the length of the individuals

Data: $i \in 1..n$: the running index

Data: $v \in \mathbb{B}$: the running index

Output: g : a new bit string of length n with random contents

```

1 begin
2    $i \leftarrow n$ 
3    $g \leftarrow ()$ 
4   while  $i > 0$  do
5     if  $\text{randomUni}[0, 1) < 0.5$  then  $v \leftarrow 0$ 
6     else  $v \leftarrow 1$ 
7      $i \leftarrow i - 1$ 
8      $g \leftarrow \text{addItem}(g, v)$ 
9   return  $g$ 
```

Creation: Permutations

The problem space \mathbb{X} can be considered to be the space $\mathbb{X} = \Pi(S)$ of all permutations of elements of the given set S . The mapping of the vectors to the permutations can be done using

$$\text{gpm}_{\Pi}(g, S) = x: x[i] = s_{g[i]} \forall i \in 0 \dots n - 1$$

The gene at locus i in the genotype $g \in G$ defines which element s from the set S should appear at position i in the permutation $x \in \mathbb{X} = \Pi(S)$.

Creation: Permutations (cont.)

If the space \mathbb{G} consists of permutations of n natural numbers, the new genotype can be created with the following algorithm.

Algorithm 29.2: $g \leftarrow \text{createGAPerm}(n)$


Input: n : the number of elements to permutate

Data: tmp : a temporary list

Data: i, j : indices into tmp and g

Output: g : the new random permutation of the numbers $0..n - 1$

```
1 begin
2    $tmp \leftarrow (0, 1, \dots, n - 2, n - 1)$ 
3    $i \leftarrow n$ 
4    $g \leftarrow ()$ 
5   while  $i > 0$  do
6      $j \leftarrow \lfloor \text{randomUni}[0, i) \rfloor$ 
7      $i \leftarrow i - 1$ 
8      $g \leftarrow \text{addItem}(g, tmp[j])$ 
9      $tmp[j] \leftarrow tmp[i]$ 
10  return  $g$ 
```



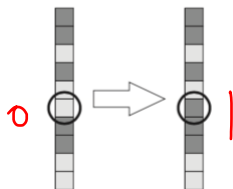
Mutation: Single-Point

Mutation is a unary reproduction

Mutation introduces small random changes to the candidate solutions.

A random modification of an allele of a gene achieves a **single-point** mutation with all points having Hamming distance of 1.

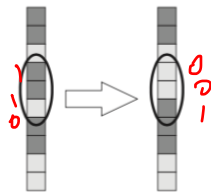
$$\text{adjacent}_{\epsilon}(g_2, g_1) = \text{dist_ham}(g_2, g_1) = 1$$



Single-gene mutation

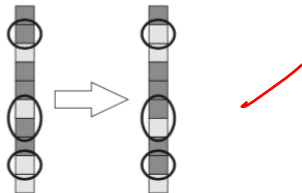
Mutation: Multi-Point

Consecutive method puts strings into adjacency which have a Hamming distance of l and where the modified bits *additionally* from a consecutive sequence.



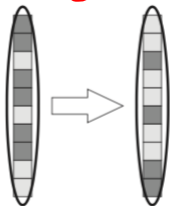
Uniform method creates a neighborhood depending on the probability $\varepsilon(l)$ based on

$$\text{adjacent}_{\varepsilon(l)}(g_2, g_1) = \text{dist_ham}(g_2, g_1) = l$$



Mutation: Complete

It is possible to modify all genes, but it discards all previously known information (a.k.a. vector mutation).

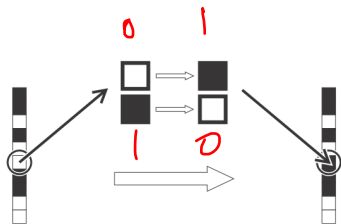


Complete mutation

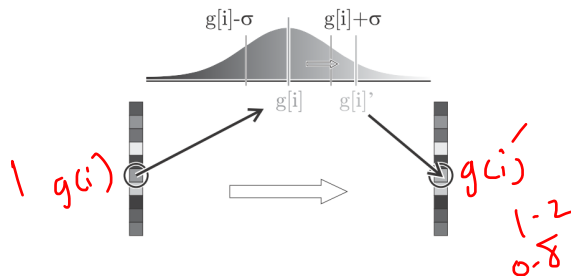
Mutation: Real Genomes

For binary-coded genomes, the mutation is called a *bit-flip mutation*.

For real-encoded genomes, modifying an element $g[i]$ can be done by replacing it with a number drawn from a normal distribution.



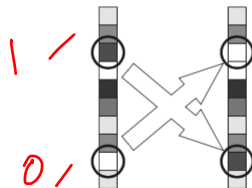
Mutation in binary genomes



Mutation in real genomes

Permutation: Unary Reproduction

The permutation operation is an alternative mutation method where the alleles of two genes are exchanged. This is only possible if the data types are similar and the change of location of a gene changes its meaning.



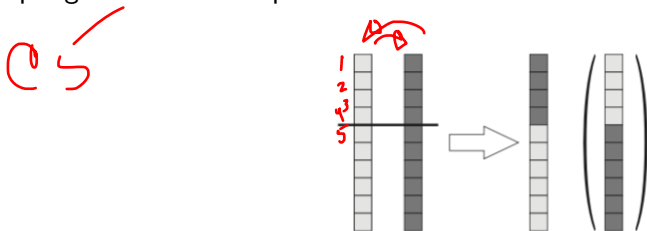
Permutation

Crossover: Binary Reproduction

Methods used to recombine two string chromosomes are called *crossover* are performed by swapping parts of two genotypes.

Single-Point Crossover (SPX, 1-PX):

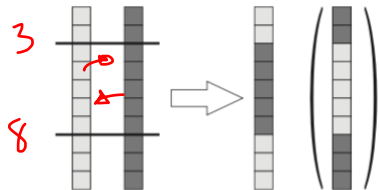
Parent chromosomes are split at a randomly determined *crossover point*. It is possible to create two offspring from the same parents.



Crossover: Binary Reproduction

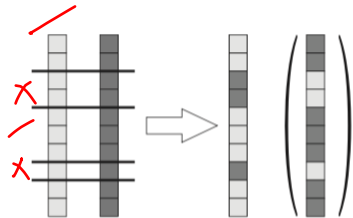
Two-Point Crossover (TPX):

Parent chromosomes are split at two randomly determined points.



Multi-Point Crossover (MPX, k-PX):

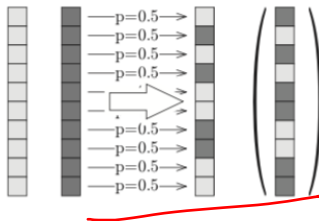
Parent chromosomes are split at k crossover points. Depending on the iteration, a portion from first parent g_{p1} or second parent g_{p2} is copied.



Crossover: Binary Reproduction

Uniform Crossover (UX):

The allelic value is chosen with the same probability from the same locus of the first parent or from the same locus of the second parent for each locus in the offspring chromosome.



Crossover: Binary Reproduction

Blending

g_1

g_2



AVG

(Weighted) Average Crossover:

For real-valued genomes the weighted average of two parents is used as an offspring. Each element of parents g_1 and g_2 is combined by a weighted average to produce an element g_{new} . The weight γ can be:

- a random number r uniformly distributed in $(0,1)$
- $0.5 \pm (0.5 * (r^n))$ to prefer points which are centered
- an approximately normally distributed random number with mean $\mu = 0.5$, again within the bounds $(0, 1)$ for each locus i

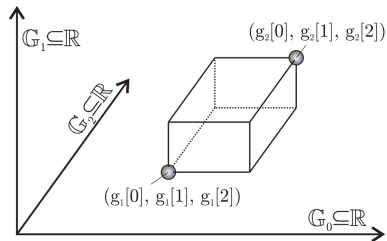
$$g_{new}[i] = \gamma g_1[i] + (1 - \gamma) g_2[i] \quad \forall i \in 1, \dots, n$$

0.9

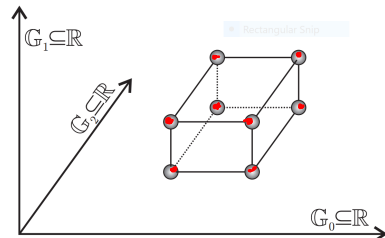
0.1

0.9

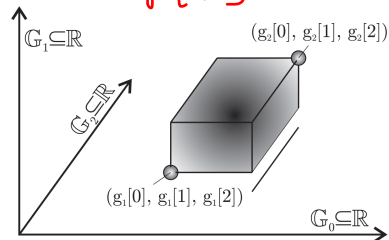
Points sampled by different crossover operators



1-PX, 2-PX, M-PX, UX
Weighted Average
Crossover, etc.



$f[0,1]$



Variable-Length String Chromosomes

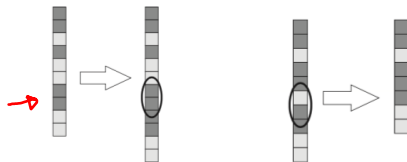
Creation: Nullary Reproduction:



Variable-length strings are created by first randomly selecting a length $l > 0$ and then creating a list of length l with random values.

Insertion and Deletion: Unary Reproduction:

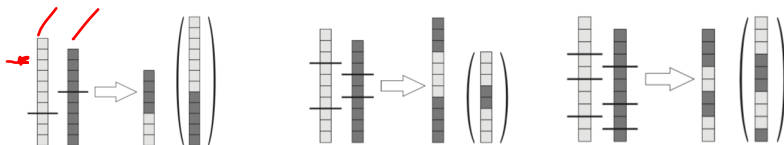
All the same unary reproduction operations (mutations) can be used for variable-string chromosomes as for fixed-length with an addition of insertion of several genes with randomly chosen alleles and deletion of some genes.



Variable-Length String Chromosomes

Crossover: Binary Reproduction:

The same crossover operations can be used for variable-length as for fixed-length strings except that the strings are no longer necessarily split at the same loci and the length of offspring can differ from the lengths of the parents.



Crossover of variable-length string chromosomes (SPX | TPX | MPX)

A simplified view of Schema Theorem



GAs work with very simple operations: string copying, substring exchange, bit (or gene) alteration – why does this work?

There is no straightforward general answer; however, for the simple genetic algorithm, there is the following theorem.

Schema theorem

Short, low order, above average schemata receive exponentially increasing trials in subsequent generations of a genetic algorithm.

Schema

Schema is a similarity template describing a subset of strings with similarities at certain string positions; consider:

Alphabet $V = \{0, 1, *\}$

Sample string *11*0** (length = 7)

Symbol * is a wildcard (don't care, 0 or 1)

Ex 1: $S = \underline{*110111} \rightarrow \underline{1110111}, \underline{0110111}$ (2 strings)

Ex 2: $S = \underline{1*1011*}$
 $\rightarrow \underline{1010110}, \underline{1010111}, \underline{1110110}, \underline{1110111}$ (4 strings)

Ex 3: $S = \underline{1***0**} \rightarrow$ (? strings)

0110 011
 -1 -00

2⁵ = 32

100

Schema description

Schema length: $d(S)$, the distance between the first and last specific string position

$$1 * 1011 * \rightarrow d = 6 - 1 = 5$$

$$* \underline{11} * \underline{0} * * \rightarrow d = 5 - 2 = 3$$

Schema order: $o(S) = l - m$, the number of not don't care positions

$$1 * 1011 * \rightarrow o = 7 - 2 = 5$$

$$\checkmark * \underline{11} * \underline{0} * * \rightarrow o = 3 \Rightarrow 7 - 4 = 3$$

The **schema theory** describes **how** each **schema** in the population **evolves** through time:

- Let $\Phi(S, g)$ be the number of instances representing schema S at generation g .
- What to expect of $\Phi(S, g + 1)$? *# Schema pop. @ $g + 1$.*

Schema pop. @ generation g .

Selection and fitness of schemata

POP — (1 — — 100)
 $\underbrace{\hspace{1cm}}_S$

Probability of selecting an individual i :

indiv. i $P_i(g) = \frac{f_i(g)}{\sum f_j(g)}, \text{ or } \frac{f_i(g)}{n f_{\text{ave}}(g)}$

indiv	Fitness
1	$f(1) \quad s_1$
2	$\vdots \quad s_2$
\vdots	\vdots
i	$\frac{f(i) \quad s_i}{\sum f_i}$

Average fitness value of the members of schema S can be defined in the following way

$$f_{\text{ave}}(S, g) = \frac{\sum_S f_i}{\Phi(S, g)}$$

where the sum goes through only the individuals covered by schema S .

$\rightarrow \frac{f_{s1} + f_{s2} + f_{si}}{3}$
 $\rightarrow \frac{\sum f_i}{n}$

Expectation for $g + 1$

S_1 1* * * 0 0 1
 S_2 x x x x x 1

$f_{avg} = \frac{2}{5} = 0.4$

Example

indiv \rightarrow all fitness
 S_1 S_2

indiv	fitness	S_1	S_2
1	0.1	0.1	0.7
*2	0.7		
*3	0.4		0.4
4	0.2	0.2	
5	0.6		

The expected value of the number of instances in schema S at generation $g + 1$ is:

$$E[\Phi(S, g + 1)] = \frac{f_{ave}(S, g)}{f_{ave}(g)} \Phi(S, g),$$

$Avg\ fitness = \frac{0.3}{2} = 0.15$
 $\rightarrow \text{Fit } S_2 = \frac{1.1}{2} = 0.55$

which means the value is **proportional** to the average fitness of those individuals in schema S at generation g and **inversely proportional** to the average fitness of all individuals at this generation.

$F(S_2) > 1 \quad \text{so } E(\Phi(S_1, g+1)) \uparrow$
 $F(S_2) < 1 \quad = \quad \downarrow$

Schema growth equation

$$f(\text{all})$$

0.4

$$f(S)$$

0.55

$$f(S) = f(\text{all}) + \beta(f(\text{all}))$$

Assume that fitness of S remains constantly above average by β , i.e.

$$E[\Phi(S, g + 1)] = \frac{f_{\text{ave}}(g) + \beta f_{\text{ave}}(g)}{f_{\text{ave}}(g)} \Phi(S, g),$$

$$f(S, g)$$

which can be simplified as

$$E[\Phi(S, g + 1)] = (1 + \beta) \Phi(S, g).$$

This means that **better**/worse schemata will receive exponentially **increasing**/decreasing number of trials in the subsequent generations.

Effects of Mutation and Crossover

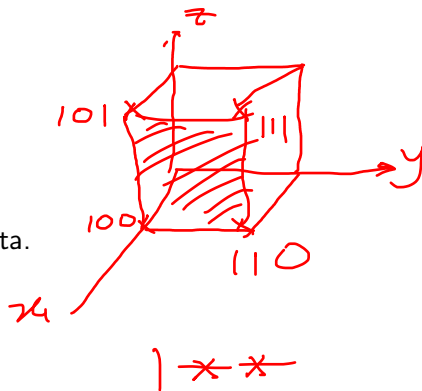
Above average schemata experience growth.

Crossover is more likely to destroy **long** schemata.

$***10***10***$ $l=11-4=7$
 $****1010****$ $9-6=3$
 Break schema rule.

Mutation is more likely to destroy **high-order** schemata.

$3***10**10**10**$ $order=6$
 $4**10*****10**$ $order=4$



Schema theorem

Short, **low-order**, **above average** schemata receive exponentially increasing trials in subsequent generations of a genetic algorithm.

GAs Applications

- Scheduling problems (e.g, job-shop scheduling, traveling salesman problem)
- Optimization of network topologies
- Resource allocation over a distributed system
- Electronic circuit design
- Aircraft design
- Game playing
- Training of fuzzy systems or artificial neural networks