STEVENS
INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY

CPE/EE 695: Applied Machine Learning

Lecture 10: Genetic Algorithms

Dr. Shucheng Yu, Associate Professor
Department of Electrical and Computer Engineering
Stevens Institute of Technology

Evolution in Biology

Organisms (animals or plants) produce a number of offspring which are almost, but not entirely, like themselves

- Variation may be due to **mutation** (random changes)
- Variation may be due to **reproduction** (offspring have some characteristics from each parent)

Some of these offspring may survive to produce offspring of their own—some won't

- The "better adapted" offspring are more likely to survive
- Over time, later generations become better and better adapted

Genetic algorithms use this same process to "evolve" better programs

Genetic Algorithms

Population of individuals

- Individual is feasible solution to problem

Each individual is characterized by a **Fitness function**

- Higher fitness is better solution

Based on their fitness, parents are selected to reproduce **offspring** for a new **generation**

- Fitter individuals have more chance to reproduce
- New generation has same size as old generation; old generation dies

Offspring has **combination** of properties of two parents

If well designed, population will **converge** to optimal solution

GA($Fitness$, $Fitness\_threshold$, p, r, m)

Fitness: A function that assigns an evaluation score, given a hypothesis.

Fitness_threshold: A threshold specifying the termination criterion.

p: The number of hypotheses to be included in the population.

r: The fraction of the population to be replaced by Crossover at each step.

m: The mutation rate.

- *Initialize population:* $P \leftarrow$ Generate p hypotheses at random
- *Evaluate:* For each h in P, compute $Fitness(h)$
- While $[\max_h Fitness(h)] < Fitness\_threshold$ do

  Create a new generation, P_S:

  1. *Select:* Probabilistically select $(1 - r)p$ members of P to add to P_S. The probability $\Pr(h_i)$ of selecting hypothesis h_i from P is given by

$$\Pr(h_i) = \frac{Fitness(h_i)}{\sum_{j=1}^{p} Fitness(h_j)}$$

  2. *Crossover:* Probabilistically select $\frac{r \cdot p}{2}$ pairs of hypotheses from P, according to $\Pr(h_i)$ given above. For each pair, $\langle h_1, h_2 \rangle$, produce two offspring by applying the Crossover operator. Add all offspring to P_s.
  3. *Mutate:* Choose m percent of the members of P_s with uniform probability. For each, invert one randomly selected bit in its representation.
  4. *Update:* $P \leftarrow P_s$.
  5. *Evaluate:* for each h in P, compute $Fitness(h)$

- Return the hypothesis from P that has the highest fitness.

Basic principles 1

Coding or **Representation**

String with all parameters

Fitness function

Parent selection

Reproduction

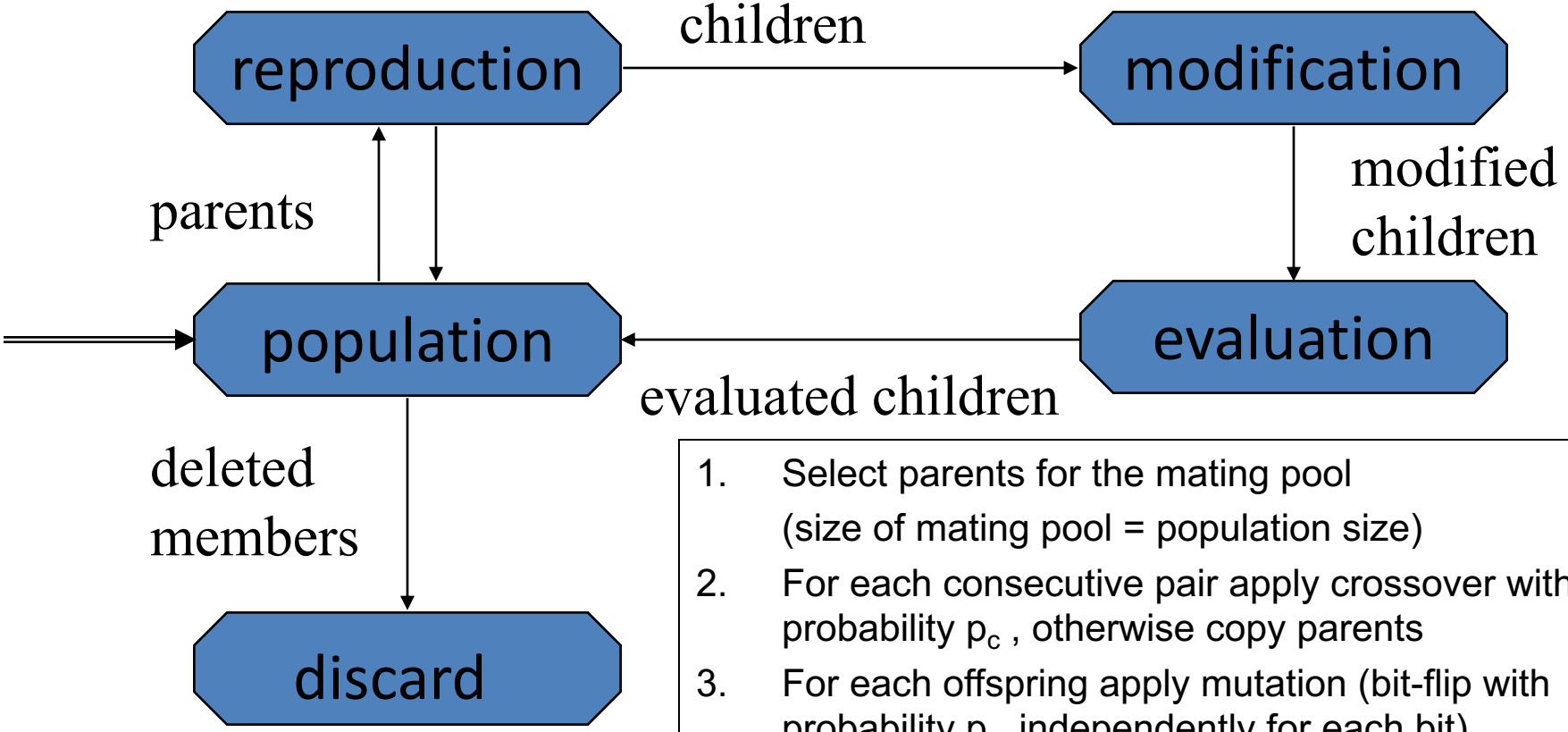
Crossover

Mutation

Convergence

When to stop

The GA Cycle of Reproduction



1. Select parents for the mating pool
   (size of mating pool = population size)
2. For each consecutive pair apply crossover with probability p_c , otherwise copy parents
3. For each offspring apply mutation (bit-flip with probability p_m independently for each bit)
4. Replace the whole population with the resulting offspring

Coding

Parameters of the solution (**genes**) are concatenated to form a string (**chromosome**)

All kind of alphabets can be used for a chromosome (numbers, characters),

Bit strings	(0101 ... 1100)
Real numbers	(43.2 -33.1 ... 0.0 89.2)
Permutations of element	(E11 E3 E7 ... E1 E15)
Lists of rules	(R1 R2 R3 ... R22 R23)
Program elements	(genetic programming)
... any data structure ...	

Order of genes on chromosome can be important

Generally many different codings for the parameters of a solution are possible

Reproduction Operators

Crossover

Two parents produce two offspring

There is a chance that the chromosomes of the two parents are copied unmodified as offspring

There is a chance that the chromosomes of the two parents are randomly recombined (crossover) to form offspring

Generally the chance of crossover is between 0.6 and 1.0

Mutation

There is a chance that a gene of a child is changed randomly

Generally the chance of mutation is low (e.g. 0.001)

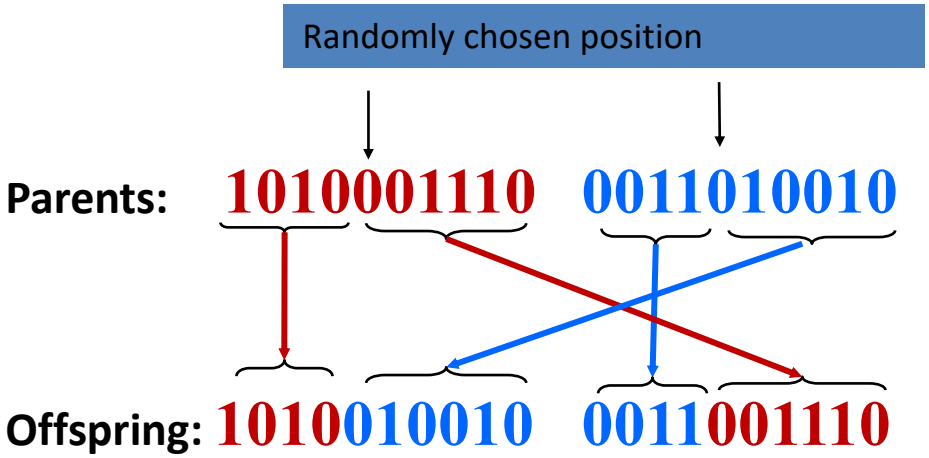
Reproduction Operators -- Crossover

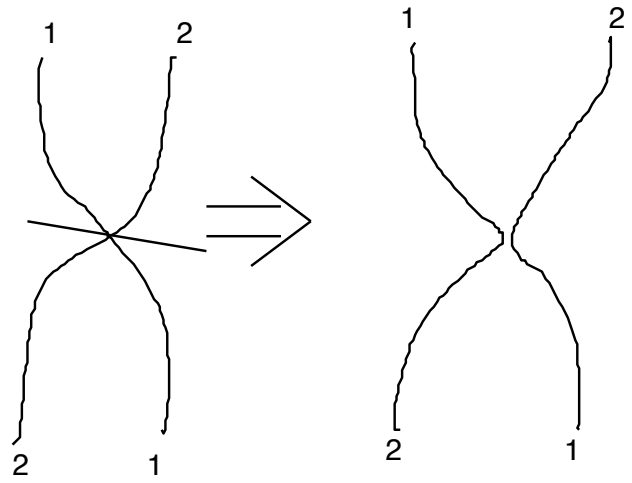
Crossover

- Generating offspring from two selected parents
  - Single point crossover
  - Two point crossover (Multi point crossover)
  - Uniform crossover

One-point crossover

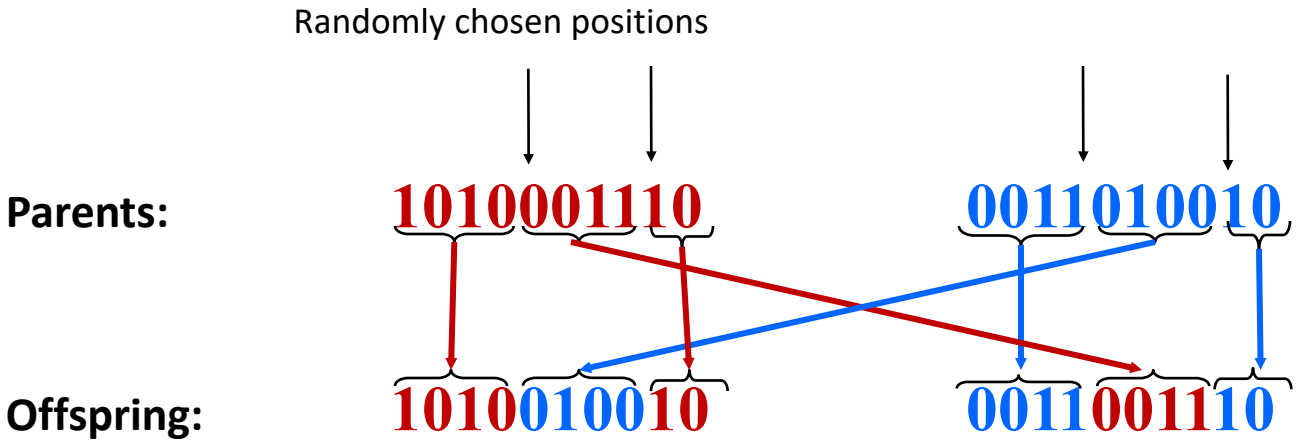
- Randomly one position in the chromosomes is chosen
- Child 1 is head of chromosome of parent 1 with tail of chromosome of parent 2
- Child 2 is head of 2 with tail of 1





Two-point crossover

- Two positions in the chromosomes are chosen randomly
- Avoids that genes at the head and genes at the tail of a chromosome are always split when recombined



Uniform crossover

A random mask is generated

The mask determines which bits are copied from one parent and which from the other parent

Bit density in mask determines how much material is taken from the other parent (takeover parameter)

Mask: **0110011000** **(Randomly generated)**

Parents: **1010001110** **0011010010**

Offspring: **0011001010** **1010010110**

Crossover operators for real valued GAs

Discrete:

each allele value in offspring *z* comes from one of its parents *(x,y)* with equal probability: $z_i = x_i$ or y_i

Could use n-point or uniform

Intermediate

exploits idea of creating children "between" parents (hence a.k.a. *arithmetic recombination*)

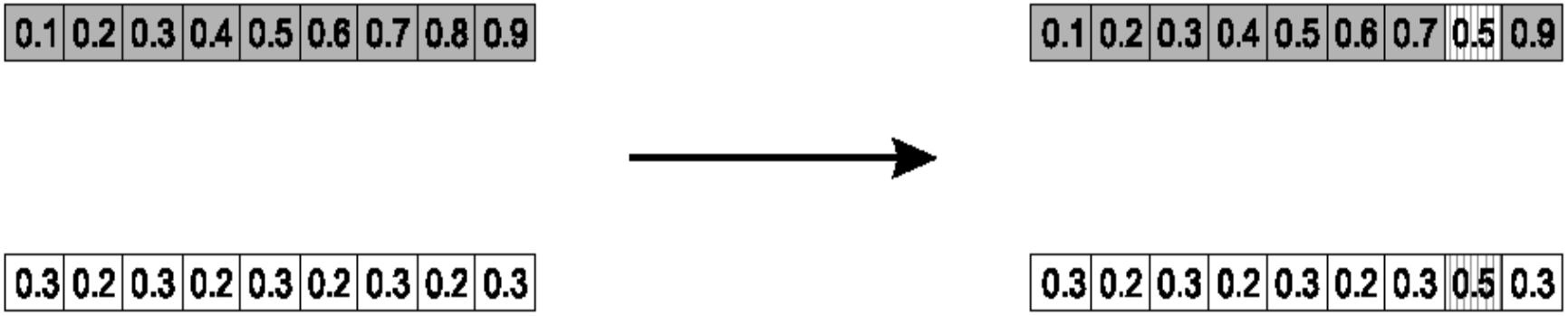
$$z_i = \alpha x_i + (1 - \alpha) y_i \quad \text{where } \alpha : 0 \leq \alpha \leq 1.$$

The parameter α can be:

- constant: uniform arithmetical crossover
- variable (e.g. depend on the age of the population)
- picked at random every time

Single arithmetic crossover

- Parents: $\langle x_1,\ldots,x_n \rangle$ and $\langle y_1,\ldots,y_n \rangle$
- Pick a single gene (k) at random,
- $child_1$ is: $\langle x_1, \ldots, x_k, \alpha \cdot y_k + (1-\alpha) \cdot x_k, \ldots, x_n \rangle$
- reverse for other child. e.g. with $\alpha = 0.5$

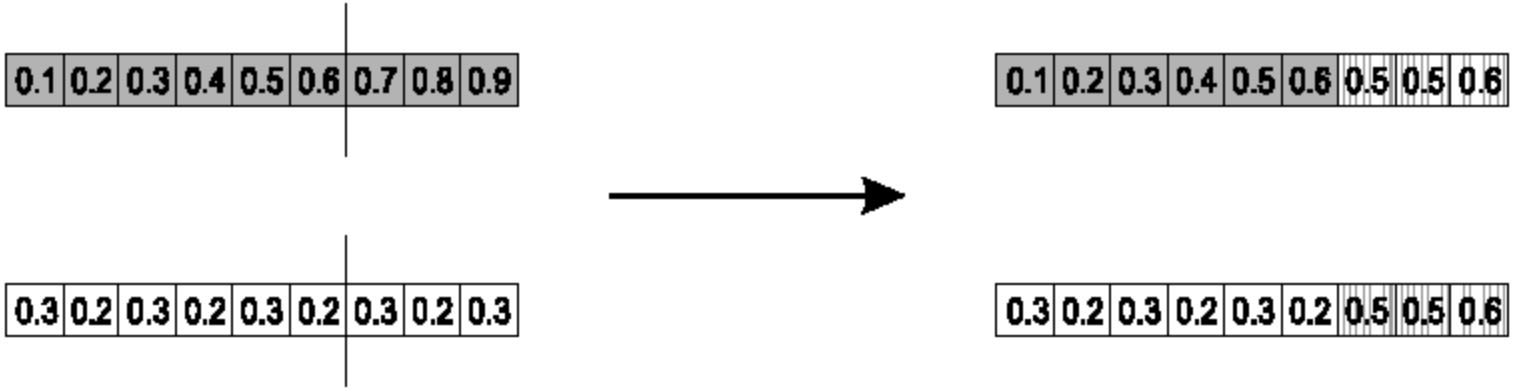


Simple arithmetic crossover

- Parents: $\langle x_1,\ldots,x_n \rangle$ and $\langle y_1,\ldots,y_n \rangle$
- Pick random gene *(k)* after this point mix values
- $child_1$ is:

$$\left\langle x_1,\ldots,x_k,\alpha\cdot y_{k+1}+(1-\alpha)\cdot x_{k+1},\ldots,\alpha\cdot y_n+(1-\alpha)\cdot x_n \right\rangle$$

- reverse for other child. e.g. with $\alpha = 0.5$

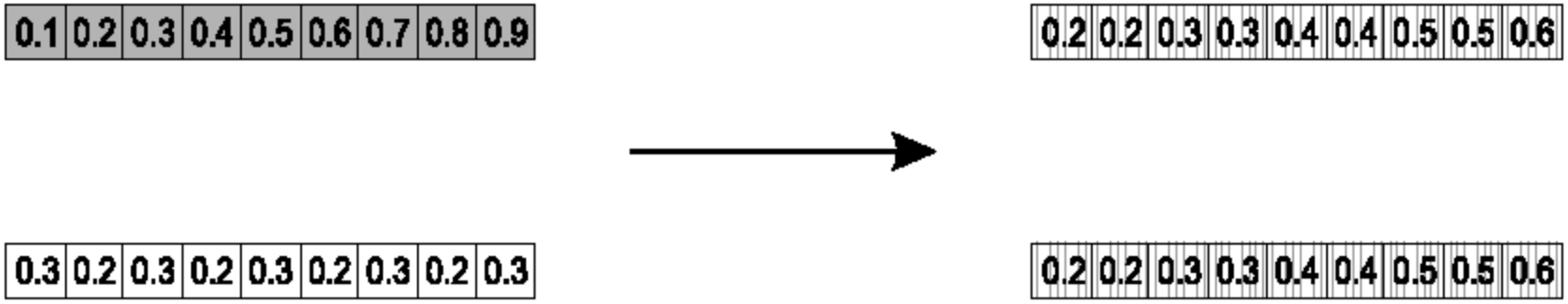


Whole arithmetic crossover

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

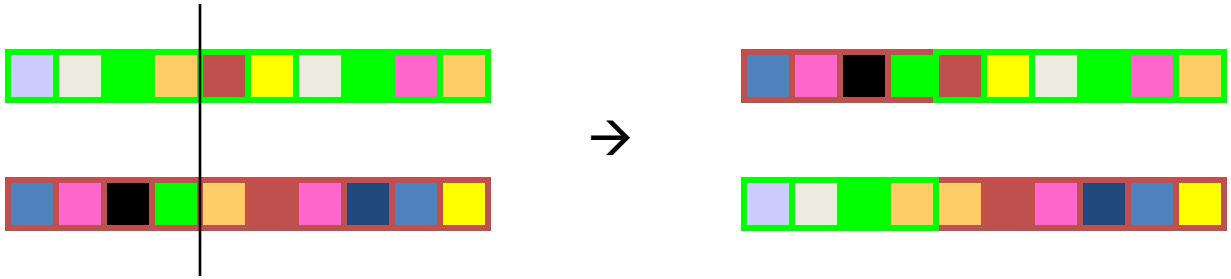
$$a \cdot \bar{x} + (1-a) \cdot \bar{y}$$

- reverse for other child. e.g. with $\alpha = 0.5$

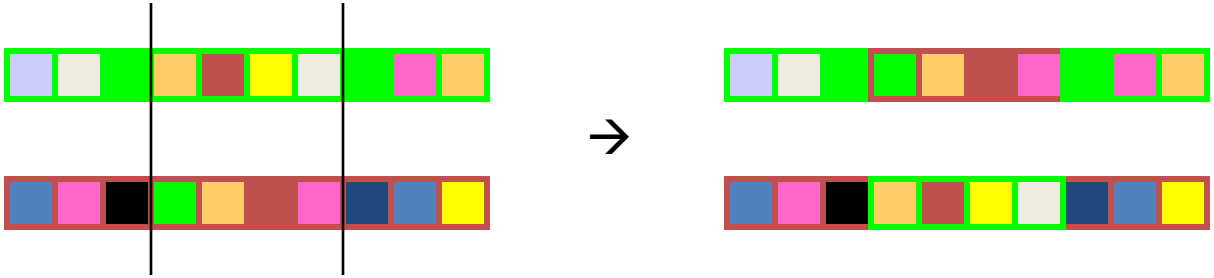


Crossover comparison

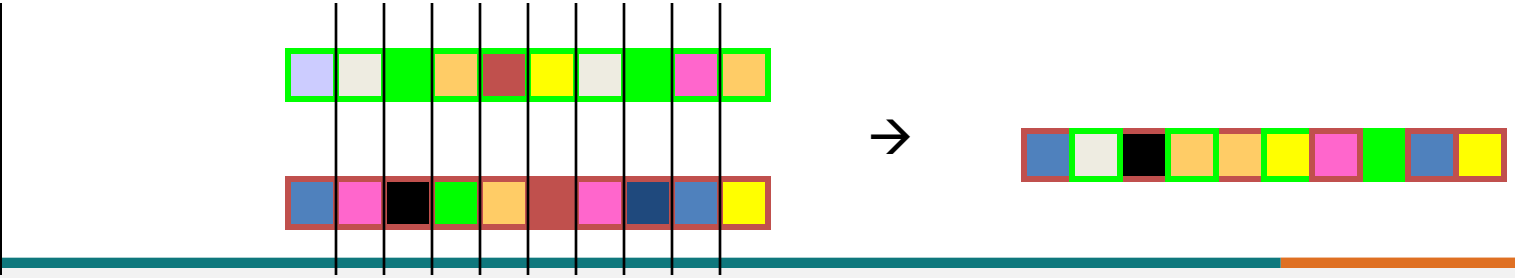
Single point crossover



- Two point crossover (Multi point crossover)



- Uniform crossover



Crossover comparison

Is uniform crossover better than single crossover point?

- Trade off between

  Exploration: Introduction of new combination of features and discover promising areas in the search space, i.e. gaining information on the problem

  Exploitation: Keep the good features in the existing solution and optimize within a promising area, i.e. using information

Problems with crossover

Depending on coding, simple crossovers can have high chance to produce illegal offspring

- E.g. in TSP with simple binary or path coding, most offspring will be illegal because not all cities will be in the offspring and some cities will be there more than once

Uniform crossover can often be modified to avoid this problem

- E.g. in TSP with simple path coding:
  - Where mask is 1, copy cities from one parent
  - Where mask is 0, choose the remaining cities in the order of the other parent

Reproduction Operators

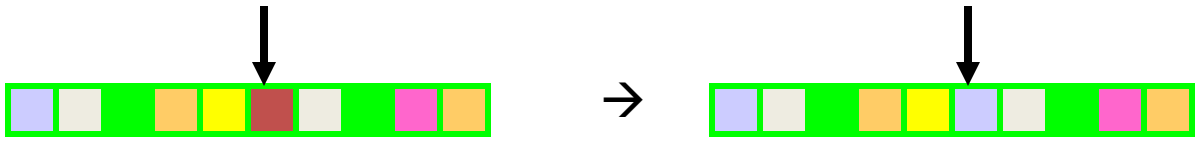
Mutation

Generating new offspring from single parent

Alter each gene independently with a probability p_m

p_m is called the mutation rate

Typically between 1/pop_size and 1/ chromosome_length



Maintaining the diversity of the individuals

Crossover can only explore the combinations of the current gene pool

Mutation can “generate” new genes

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 is possible, xover-only would not work

Crossover OR mutation? (cont'd)

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 (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

Reproduction Operators

Control parameters: **population size, crossover and mutation probability**

- Problem specific
- Increase population size
  - Increase diversity and computation time for each generation
- Increase crossover probability
  - Increase the opportunity for recombination but also disruption of good combination
- Increase mutation probability
  - Closer to randomly search
  - Help to introduce new gene or reintroduce the lost gene

Varies the population

Usually using crossover operators to recombine the genes to generate the new population, then using mutation operators on the new population

Operators: Selection

Main idea: better individuals get higher chance

Chances proportional to fitness

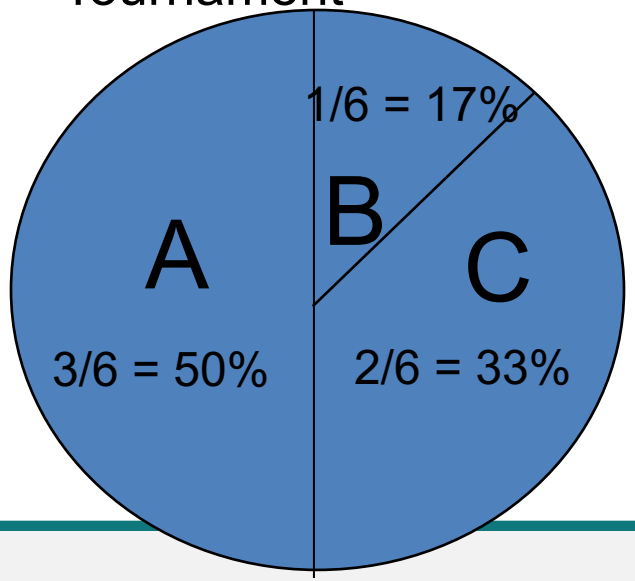
Implementation:

Roulette wheel selection (Fitness proportionate selection)

Assign to each individual a part of the roulette wheel

Spin the wheel n times to select n individuals

Tournament



fitness(A) = 3

fitness(B) = 1

fitness(C) = 2

Tournament

Binary tournament

Two individuals are randomly chosen; the fitter of the two is selected as a parent

Probabilistic binary tournament

Two individuals are randomly chosen; with a chance p, $0.5<p<1$, the fitter of the two is selected as a parent

Larger tournaments

n individuals are randomly chosen; the fittest one is selected as a parent

By changing n and/or p, the GA can be adjusted dynamically

Fitness Function

Purpose

Parent selection

Measure for convergence

For Steady state: Selection of individuals to die

Should reflect the value of the chromosome in some “real” way

Next to coding the most critical part of a GA

A simple example

Suppose your “organisms” are 32-bit computer words

You want a string in which all the bits are ones

Here’s how you can do it:

- Create 100 randomly generated computer words
- Repeatedly do the following:
  - Count the 1 bits in each word
  - Exit if any of the words have all 32 bits set to 1
  - Keep the ten words that have the most 1s (discard the rest)
  - From each word, generate 9 new words as follows:
    - Pick a random bit in the word and change it

Note that this procedure does not guarantee that the next “generation” will have more 1 bits, but it’s likely

A simple example – GA solution

Suppose your “organisms” are 32-bit computer words, and you want a string in which all the bits are ones

Here’s how you can do it:

- Create 100 randomly generated computer words
- Repeatedly do the following:
  - Count the 1 bits in each word
  - Exit if any of the words have all 32 bits set to 1
  - Keep the ten words that have the most 1s (discard the rest)
  - From each word, generate 9 new words as follows:
    - Choose one of the other words
    - Take the first half of this word and combine it with the second half of the other word

The example continued

Half from one, half from the other:
0110 1001 0100 1110 1010 1101 1011 0101
1101 0100 0101 1010 1011 0100 1010 0101
0110 1001 0100 1110 1011 0100 1010 0101

Or we might choose “genes” (bits) randomly:
0110 1001 0100 1110 1010 1101 1011 0101
1101 0100 0101 1010 1011 0100 1010 0101
0100 0101 0100 1010 1010 1100 1011 0101

Or we might consider a “gene” to be a larger unit:
0110 1001 0100 1110 1010 1101 1011 0101
1101 0100 0101 1010 1011 0100 1010 0101
1101 1001 0101 1010 1010 1101 1010 0101

Comparison of simple examples

In the simple example (trying to get all 1s):

- The crossover approach, if it succeeds, is likely to succeed much faster
  - Because up to half of the bits change each time, not just one bit
- However, with no mutation, it may not succeed at all
  - By pure bad luck, maybe *none* of the first (randomly generated) words have (say) bit 17 set to 1
    - Then there is no way a 1 could ever occur in this position
  - Another problem is lack of ***genetic diversity***
    - Maybe some of the first generation did have bit 17 set to 1, but none of them were selected for the second generation

The best technique *in general* turns out to be crossover with a *small* probability of mutation

Directed evolution

Notice that, in the previous examples, we formed the child organisms *randomly*

- We did not try to choose the "best" genes from each parent
- This is how natural (biological) evolution works
  - Biological evolution is *not directed*—there is no "goal"
- Genetic algorithms use biology as *inspiration,* not as a set of rules to be slavishly followed

For trying to get a word of all 1s, there is an obvious measure of a "good" gene

- But that's mostly because it's a silly example
- It's much harder to detect a "good gene" in the curve-fitting problem, harder still in almost any "real use" of a genetic algorithm

A realistic example

Suppose you have a large number of (x, y) data points

- For example, (1.0, 4.1), (3.1, 9.5), (-5.2, 8.6), …

You would like to fit a polynomial (of up to degree 5) through these data points

- That is, you want a formula $y = ax^5 + bx^4 + cx^3 + dx^2 + ex + f$ that gives you a reasonably good fit to the actual data
- Here's the usual way to compute goodness of fit:
  - Compute the sum of $(\text{actual } y - \text{predicted } y)^2$ for all the data points
  - The lowest sum represents the best fit

There are some standard curve fitting techniques, but let's assume you don't know about them

You can use a genetic algorithm to find a "pretty good" solution

A realistic example

Your formula is $y = ax^5 + bx^4 + cx^3 + dx^2 + ex + f$

Your "genes" are a, b, c, d, e, and f

Your "chromosome" is the array [a, b, c, d, e, f]

Your evaluation function for *one* array is:

- For *every* actual data point (x, y),
  - Compute $\acute{y} = ax^5 + bx^4 + cx^3 + dx^2 + ex + f$
  - Find the sum of $(y - \acute{y})^2$ over all x
  - The sum is your measure of "badness" (larger numbers are worse)

Example: For [0, 0, 0, 2, 3, 5] and the data points (1, 12) and (2, 22):

- $\acute{y} = 0x^5 + 0x^4 + 0x^3 + 2x^2 + 3x + 5$ is $2 + 3 + 5 = 10$ when x is 1
- $\acute{y} = 0x^5 + 0x^4 + 0x^3 + 2x^2 + 3x + 5$ is $8 + 6 + 5 = 19$ when x is 2
- $(12 - 10)^2 + (22 - 19)^2 = 2^2 + 3^2 = 13$
- If these are the only two data points, the "badness" of [0, 0, 0, 2, 3, 5] is 13

A realistic example

Your algorithm might be as follows:

- Create 100 six-element arrays of random numbers
- Repeat 500 times (or any other number):
  - For each of the 100 arrays, compute its badness (using all data points)
  - Keep the ten best arrays (discard the other 90)
  - From each array you keep, generate nine new arrays as follows:
    - Pick a random element of the six
    - Pick a random floating-point number between 0.0 and 2.0
    - Multiply the random element of the array by the random floating-point number
- After all 500 trials, pick the best array as your final answer

A realistic example – GA solution

Your formula is $y = ax^5 + bx^4 + cx^3 + dx^2 + ex + f$

Your "genes" are a, b, c, d, e, and f

Your "chromosome" is the array [a, b, c, d, e, f]

What's the best way to combine two chromosomes into one?

- You could choose the first half of one and the second half of the other: [a, b, c, d, e, f]
- You could choose genes randomly: [a, b, c, d, e, f]
- You could compute "gene averages:"
  [(a+a)/2, (b+b)/2, (c+c)/2, (d+d)/2, (e+e)/2,(f+f)/2]

Another Example :Travelling Salesman Problem

Problem:

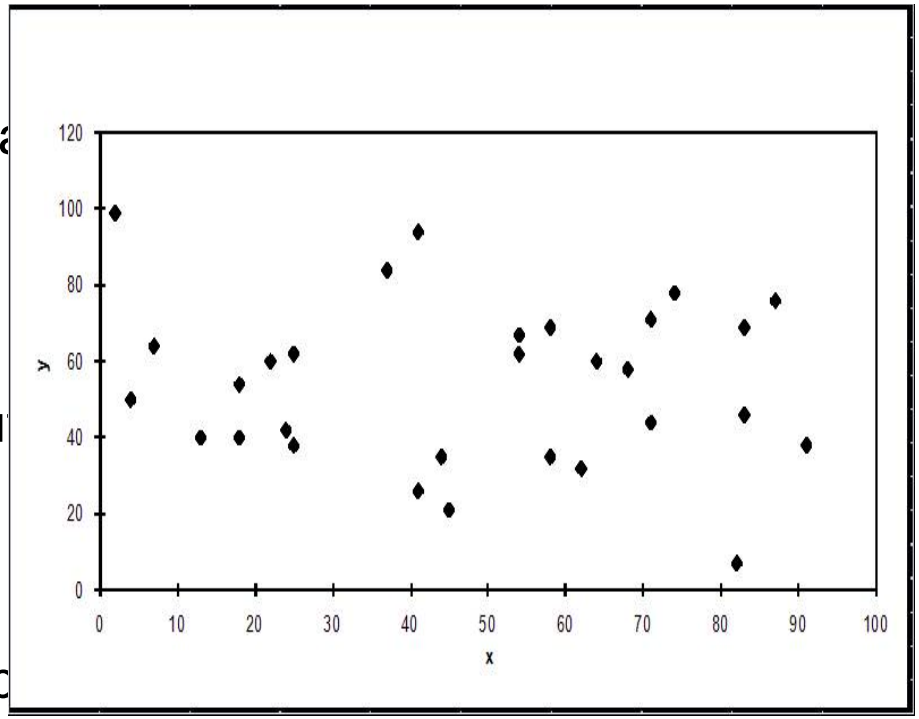
- •Given n cities
- •Find a complete tour with minima

Encoding:

- •Label the cities 1, 2, … , *n*
- •One complete tour is one permu
  are OK)

Search space is BIG:

for 30 cities there are 30! ≈ 10^{32} p



Permutation Representations

Ordering/sequencing problems form a special type

Task is (or can be solved by) arranging some objects in a certain order

Example: sort algorithm: important thing is which elements occur before others (order)

Example: Travelling Salesman Problem (TSP) : important thing is which elements occur next to each other (adjacency)

These problems are generally expressed as a permutation:

if there are *n* variables then the representation is as a list of *n* integers, each of which occurs exactly once

Mutation operators for permutations

Normal mutation operators lead to inadmissible solutions

> e.g. bit-wise mutation : let gene *i* have value *j* changing to some other value *k* would mean that *k* occurred twice and *j* no longer occurred

Therefore must change at least two values

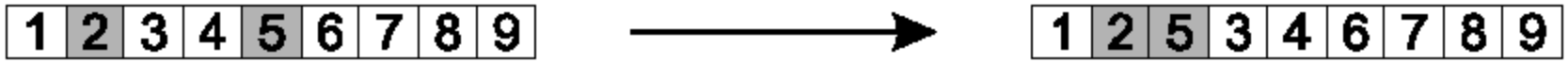
Mutation parameter now reflects the probability that some operator is applied once to the whole string, rather than individually in each position

Insert Mutation for permutations

Pick two allele values at random

Move the second to follow the first, shifting the rest along to accommodate

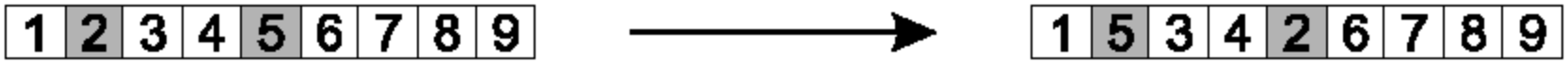
Note that this preserves most of the order and the adjacent information



Swap mutation for permutations

Pick two alleles at random and swap their positions

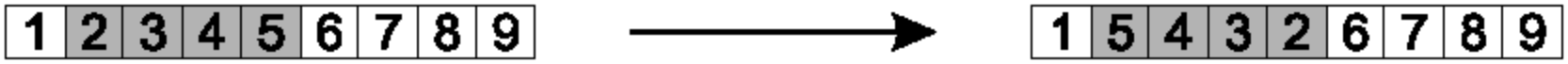
Preserves most of adjacent information (4 links broken), disrupts order more



Inversion mutation for permutations

Pick two alleles at random and then invert the substring between them.

Preserves most adjacent information (only breaks two links) but disruptive of order information

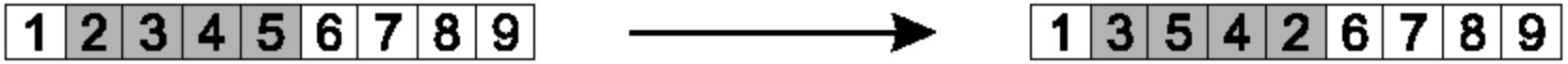


Scramble mutation for permutations

Pick a subset of genes at random

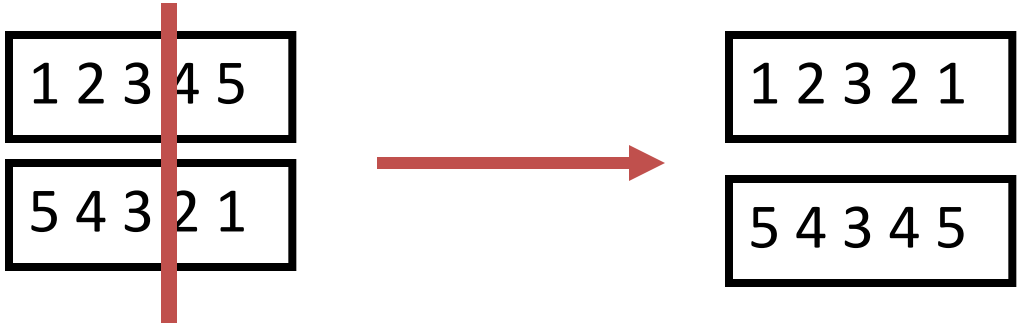
Randomly rearrange the alleles in those positions

(note subset does not have to be contiguous)



Crossover operators for permutations

"Normal" crossover operators will often lead to inadmissible solutions



Many specialised operators have been devised which focus on combining order or adjacent information from the two parents

Order 1 crossover

Idea is to preserve relative order that elements occur

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:

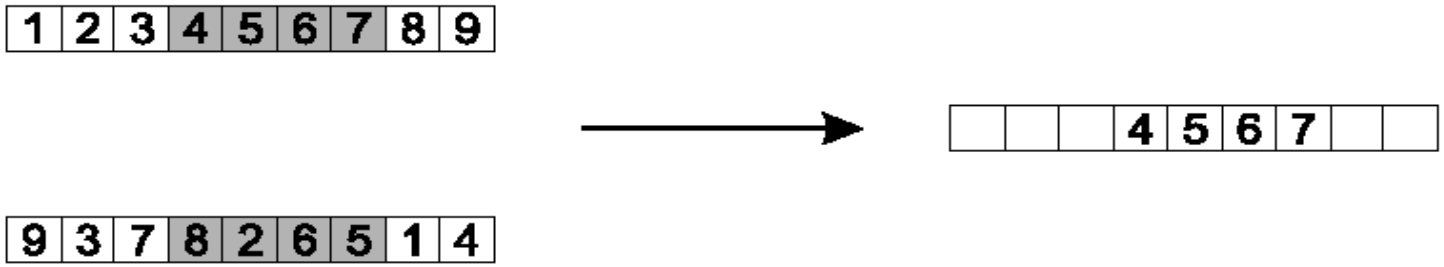
   starting right from cut point of the copied part,

   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

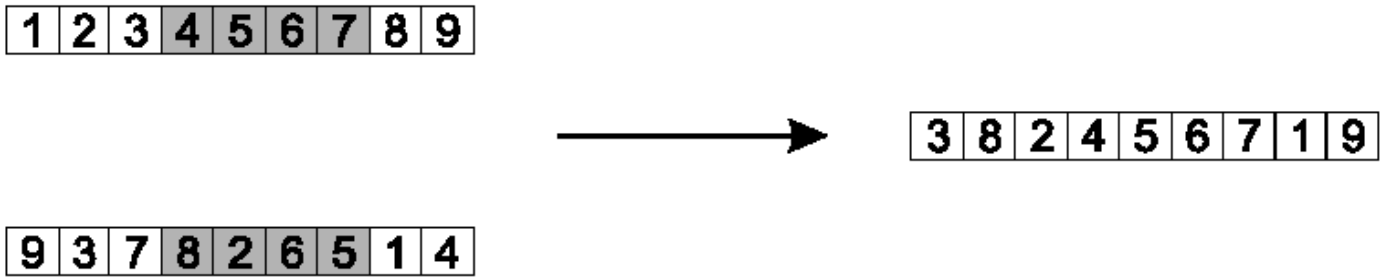
Copy randomly selected set from first parent



Rest for the first parent: 1,2,3,8,9

Order for second parent start from right of cut point:
1,4,9,3,7,8,2,6,5

Copy rest from first parent follow second parent order 1,9,3,8,2



Partially Mapped Crossover (PMX)

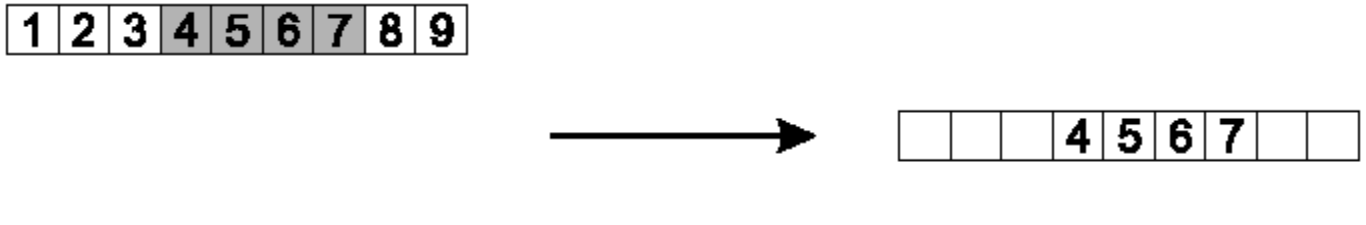
Informal procedure for parents P1 and P2:

1. Choose random segment and copy it from P1
2. Starting from the first crossover point look for elements in that segment of P2 that have not been copied
3. For each of these *i* look in the offspring to see what element *j* has been copied in its place from P1
4. 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)
5. 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
6. Having dealt with the elements from the crossover segment, the rest of the offspring can be filled from P2.

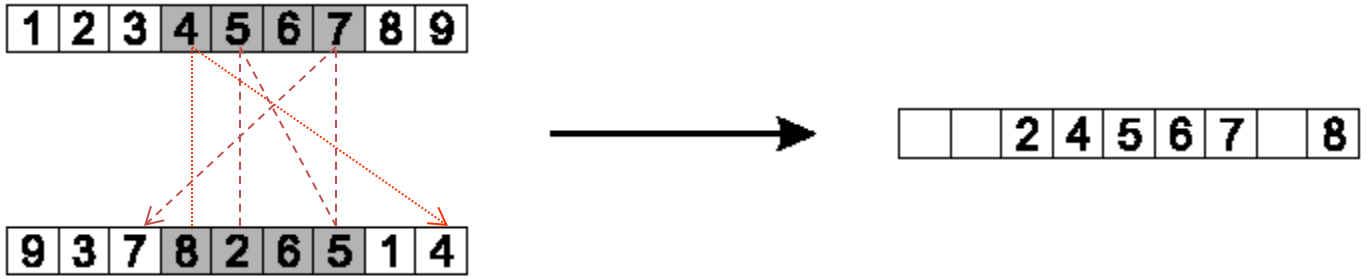
Second child is created analogously

PMX example

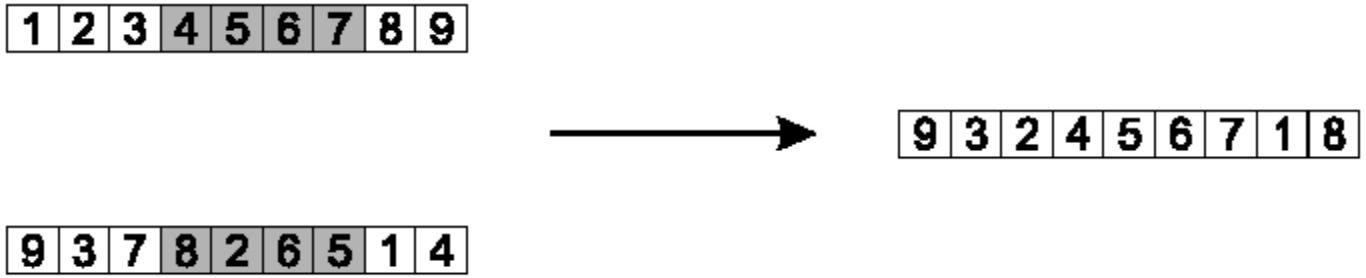
Step 1



Step 2



Step 3



Cycle crossover

Basic idea:

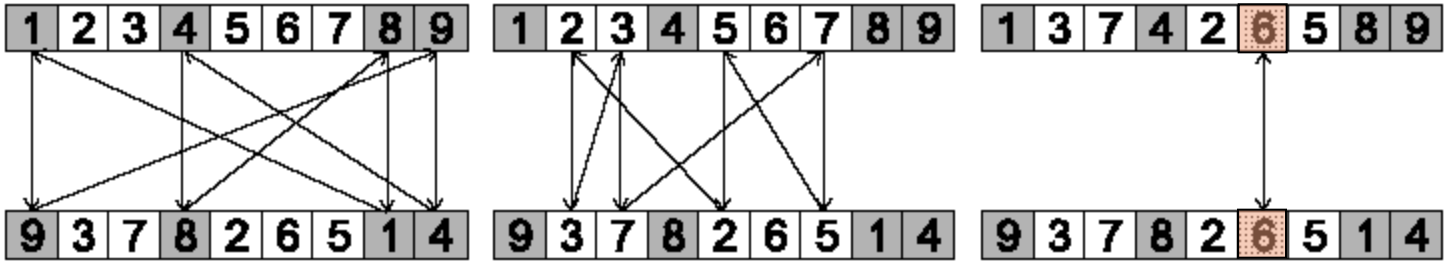
Each allele comes from one parent *together with its position*.

Informal procedure:

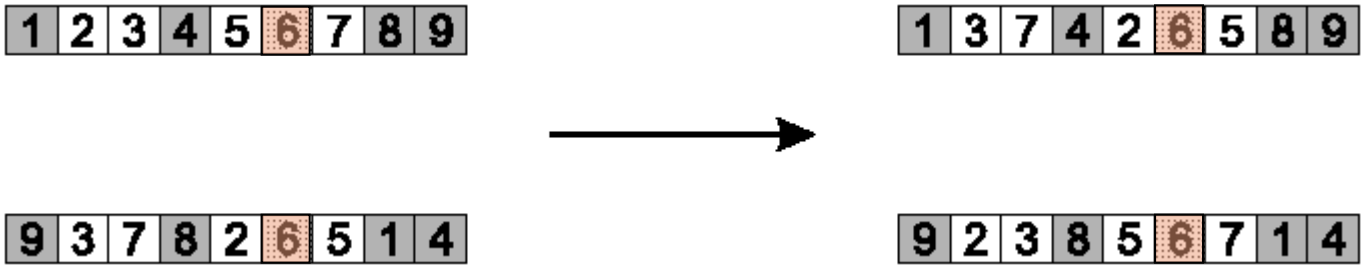
1. Make a cycle of alleles from P1 in the following way.
    (a) Start with the first allele of P1.
    (b) Look at the allele at the *same position* in P2.
    (c) Go to the position with the *same allele* in P1.
    (d) Add this allele to the cycle.
    (e) Repeat step b through d until you arrive 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.
3. Take next cycle from second parent

Cycle crossover example

Step 1: identify cycles



Step 2: copy alternate cycles into offspring



Benefits of Genetic Algorithms

Concept is easy to understand

Modular separate from application

Supports multi-objective optimization

Good for “noisy” environments

Always an answer; answer gets better with time

Inherently parallel; easily distributed

Benefits of Genetic Algorithms (cont.)

Many ways to speed up and improve a GA-based application as knowledge about problem domain is gained

Easy to exploit previous or alternate solutions

Flexible building blocks for hybrid applications

Substantial history and range of use

When to Use a GA

Alternate solutions are too slow or overly complicated

Need an exploratory tool to examine new approaches

Problem is similar to one that has already been successfully solved by using a GA

Want to hybridize with an existing solution

Benefits of the GA technology meet key problem requirements

Some GA Application Types

Domain	Application Types
Control	gas pipeline, pole balancing, missile evasion
Design	semiconductor layout, aircraft design, keyboard configuration, communication networks
Scheduling	manufacturing, facility scheduling, resource allocation
Robotics	trajectory planning
Machine Learning	designing neural networks, improving classification algorithms, classifier systems
Signal Processing	filter design
Game Playing	poker, checkers, prisoner's dilemma
Combinatorial Optimization	set covering, travelling salesman, routing, bin packing, graph colouring and partitioning

Issues for GA Practitioners

Choosing basic implementation issues:

- representation
- population size, mutation rate, ...
- selection, deletion policies
- crossover, mutation operators

Termination Criteria

Performance, scalability

Solution is only as good as the evaluation function (often hardest part)

Acknowledgement

Part of the slide materials were based on Dr. Rong Duan's Fall 2016 course CPE/EE 695A Applied Machine Learning at Stevens Institute of Technology.

Reference

The lecture notes in this lecture are based on the following textbooks:

T. M. Mitchell, Machine Learning, McGraw Hill, 1997. ISBN: 978-0-07-042807-2

STEVENS

INSTITUTE *of* TECHNOLOGY

THE INNOVATION UNIVERSITY®

stevens.edu