Genetic Algorithms

Part 4: The Component of Genetic Algorithms

Fall 2009

Instructor: Dr. Masoud Yaghini

Outline

- Representation of Individuals
- Mutation
- Recombination
- Population Models
- Parent Selection
- Survivor Selection
- Glossary
- References

Representation of Individuals

Representation of Individuals

- Depends on the problem
- The most used encodings:
  - Binary representation
  - Integer representation
  - Real-valued representation
  - Permutations representation

Binary Representations

- simplest and most common
- chromosome: string of bits
  - genes: 0 / 1
- **Example:** binary representation of an integer

  3: 00011

  15: 01111

  16: 10000

Binary Representation

- For those problem concerning **Boolean** decision variables, the genotype-phenotype mapping is natural
  - Example: knapsack problem
- Many optimization problems involve integer or real numbers and bit string can be used to encode these numbers

Mapping Integer Values on Bit Strings

- Integer values can be also binary coded:
  - $x = 5 \rightarrow 00101$

genotype	0	0	1	0	1
mapping	2^4	2^3	2^2	2^1	2^0
	16	8	4	2	1
phenotype	0*16+0*8+1*4+0*2+1*1 = 5				

Mapping real values on bit strings

- Real values can be also binary coded
- $z \in [x, y] \subseteq \mathcal{R}$ 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 $\mathbf{2^L}$ values out of infinite are represented
  - **L** determines possible maximum precision of solution
  - High precision → long chromosomes (slow evolution)

Mapping real values on bit strings

- **Example:**
  - $z \in [1, 10]$
  - We use 8 bits to represent real values in the domain [1, 10] , i.e., we use 256 numbers

    $1 \rightarrow 00000000$ (0)

    $10 \rightarrow 11111111$ (255)
  - Convert 00111100, to real value:

$1 + (9 / 255) * [(0 * 2^0) + (0 * 2^1) + (1 * 2^2) + (1 * 2^3) + (1 * 2^4) + (1 * 2^5) + (0 * 2^6) + (0 * 2^7)] = 3.12$

  - Convert n = 3.14 to bit string:

$n = 3.14 \rightarrow (3.14-1)*255/(10-1) = 60 \rightarrow 0011\ 1100$

$0011\ 1100 \rightarrow 3.12$ (better precision requires more bits)

Binary Representations

- **Hamming distance:**
  - Number of bits that have to be changed to map one string into another one
  - *E.g. 000 and 001 → distance = 1*
- Problem: **Hamming distance** between consecutive integers may be > 1
- example: 5 bit binary representation

  14: 01110 15: 01111 16: 10000
  - Probability of changing 15 into 16 by independent bit flips (mutation) is not same as changing it into 14!

Binary encoding problem

- **Remember**: small changes in genotype should cause small changes in phenotype

- **Gray coding** solves problem.

Binary Representation

- **Binary** coding of 0-7 :

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

- Hamming distance, e.g.:
  - 000 (0) and 001 (1)
    - Distance = 1 (optimal)
  - 011 (3) and 100 (4)
    - Distance = 3 (max possible)

Binary Representation

- **Gray** coding of 0-7:
  - Gray coding is a representation that ensures that consecutive integers always have Hamming distance one.
  - Gray coding is a mapping that means that small changes in the genotype cause small changes in the phenotype (unlike binary coding).

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

Binary Representation

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

- Covert Binary to Gray:
  - G[i]= B[i] : for largest i
  - G[i]= XOR(B[i+1], B[i]) : for i < largest i
- Convert Gray to Binary
  - B[i]= G[i] : if for largest i
  - B[i] = XOR(B[i+1], G[i]) : for i < largest i

- Denotes:

XOR(0, 0) = 0, XOR(0, 1) = 1,
XOR(1, 0) =1, XOR(1, 1) = 0

Binary Representation

- Nowadays it is generally accepted that it is better to encode numerical variables **directly** as:
  - Integers
  - Floating point variables

Integer representations

- Some problems naturally have **integer variables**
  - e.g. for optimization of a function with integer variables
- Values may be
  - unrestricted (all integers)
  - restricted to a finite set
    - e.g. {0,1,2,3}
    - e.g. {North,East,South,West}

Integer representations

- Any natural relations between possible values?
  - obvious for **ordinal attributes**
    - 2 is more like 3 than it is 389
    - small < medium < large
    - hot > mild > cool
  - maybe no natural ordering for **cardinal attributes**
    - e.g. set of compass points
    - e.g. employee ID

Real-valued Representation

- Many problems occur as real valued problems, e.g. continuous parameter optimization
  - Example: a chromosome can be a pair (x, y)
- Vector of real values
  - floating point numbers
- Genotype for solution becomes the vector $<x_1, x_2, \ldots, x_k>$ with $x_i \in \Re$

Permutation Representations

- Deciding on sequence of events
  - most natural representation is **permutation** of a set of integers
- In ordinary GA numbers may occur more than once on chromosome
  - invalid permutations!
- New variation operators needed

Permutation Representations

- Two classes of problems
  - based on **order of events**
    - e.g. scheduling of jobs
      - **Job-Shop Scheduling Problem**
  - based on **adjacencies**
    - e.g. **Travelling Salesperson Problem (TSP)**
      - finding a complete tour of minimal length between n cities, visiting each city only once

Permutation Representations

- **Two ways to encode a permutation**
  - *i*th element represents event that happens in that location in a sequence
  - value of *i*th element denotes position in sequence in which *i*th event occurs

- **Example (TSP):** 4 cities A,B,C,D and permutation [3,1, 2,4] denotes the tours:
  - **first encoding type:** [C→A→ B→ D]
  - **second encoding type:** [B→C→ A→ D]

Mutation

Mutation

- **Mutation** is a variation operator
- Create one offspring from one parent
- Occurs at a mutation rate: p_m
  - behaviour of a GA depends on p_m

Bitwise Mutation

- flips bits
  - 0→1 and 1→0
- setting of p_m depends on nature of problem, typically between:
  - 1 / pop_size
  - 1 / chromosome_length

Mutation for Integer Representation

- For integers there are two principal forms of mutation:
  - **Random resetting**
  - **Creep Mutation**

- Both of them, mutate each gene independently with user-defined probability p_m

Random Resetting

- A new value is chosen with from the set of permissible integer values
- Most suitable for **cardinal attributes**

Creep Mutation

- Add small (positive / negative) integer to gene value
  - random value
  - sampled from a distribution
    - symmetric around 0
    - with higher probability of small changes
- Designed for **ordinal attributes**
- **Step size** is important
  - controlled by parameters
  - setting of parameters important

Mutation for Floating-Point Representation

- Allele values come from a **continuous distribution**
- Change allele values randomly within its domain
  - upper and lower boundaries, U_i and L_i respectively

$$< x_1, x_2, ..., x_n > \rightarrow < x_1', x_2', ..., x_n' >$$
$$where\ x_i, x_i' \in [L_i, U_i]$$

Mutation for Floating-Point Representation

- According to the probability distribution from which the new gene values are drawn, There are two types of floating-point mutation:
  - **Uniform Mutation**
  - **Non-Uniform Mutation with a Fixed Distribution**

Uniform Mutation

- Values of x_i drawn uniformly randomly from the $[L_i,U_i]$, Similar to
  - bit flipping for binary representations
  - random resetting for integer representations
- usually used with positionwise mutation probability

Non-Uniform Mutation with a Fixed Distribution

- Most common form
- Similar to creep mutation for integer representations
- Add an amount to gene value
- Amount randomly drawn from a distribution

Non-Uniform Mutation with a Fixed Distribution

- Gaussian distribution (normal distribution)
  - with mean 0
  - user-specified standard deviation
  - may have to adjust to interval $[L_i, U_i]$
  - 2/3 of samples lie within one standard deviation of mean ($-\sigma$ to $+\sigma$)
  - most changes small but probability of very large changes > 0

Non-Uniform Mutation with a Fixed Distribution

- Usually applied to each gene with probability 1
- p_m used to determine standard deviation of distribution
  - determines probability distribution of size of steps taken

Mutation for Permutation Representations

- It is not possible to consider genes independently
- Move alleles around in genome
  - Therefore must change at least two values
- Mutation probability now shows the probability that mutation operator is applied once to the **whole string**, **rather than** individually in **each position**

Mutation for Permutation Representations

- There four types of mutation operators for permutation:
  - **Swap Mutation**
  - **Insert Mutation**
  - **Scramble Mutation**
  - **Inversion Mutation**

Swap Mutation

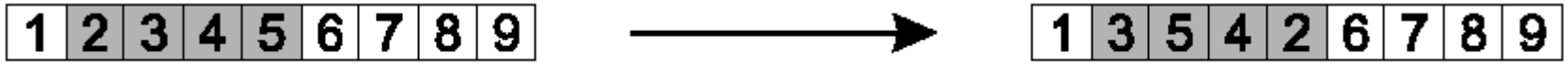
- Pick two alleles at random and swap their positions
- Preserves most of adjacency information (4 links broken), disrupts order more

Insert Mutation

- Pick **two allele** values at random
- **Move the second** to follow the first, shifting the rest along to make room
- Note that this preserves most of the order and the adjacency information

Scramble Mutation

- Pick a subset of genes at random
- Randomly rearrange the alleles in those positions



Inversion Mutation

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

Recombination

Recombination

- **Recombination** is the process for creating new individual
  - two or more parents
- Term used interchangably with **crossover**
  - Crossover mostly refers to 2 parents
- This is the primary mechanism for creating diversity.
- Crossover rate p_c
  - typically in range [0.5,1.0]
  - acts on parent pair

Recombination

- Two parents selected randomly
- a random variable drawn from [0,1)
- If **value < p_c**
  - two offspring created through recombination
- else
  - two offspring created asexually (copy of parents)

Recombination

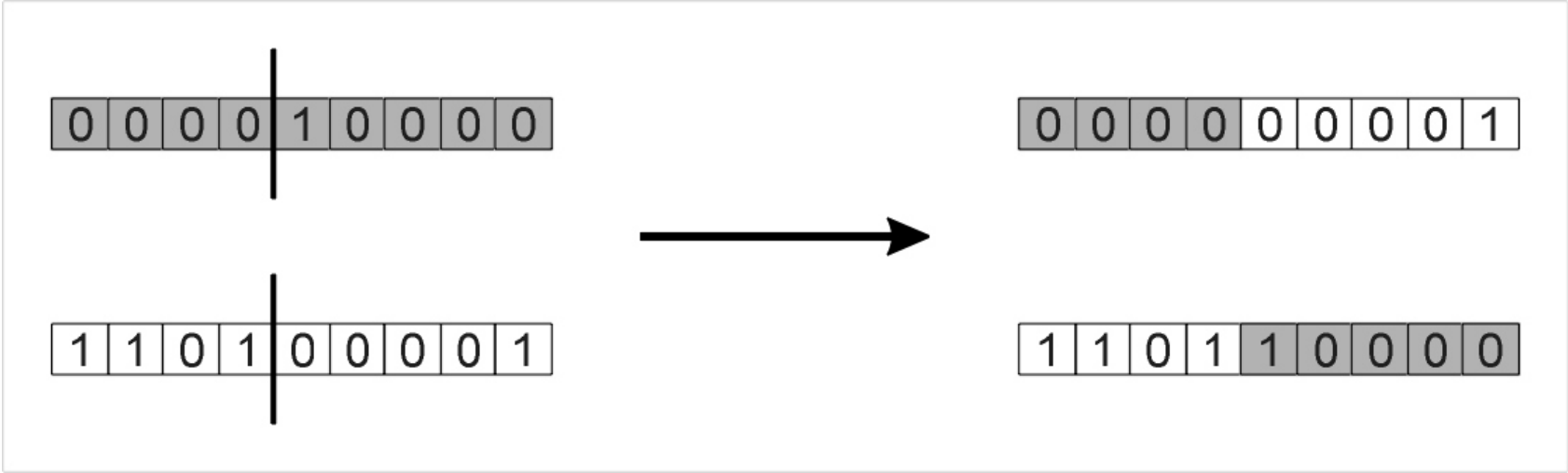
- **Crossing** over vs. **mutation** probability
  - The mutation probability p_m controls how parts of the chromosome are changed independently
  - The crossover probability p_c determines the chance that a chosen pair of parents undergoes this operator.

Recombination for Binary Representation

- There are three standard forms of recombination for binary representation:
  - **One-Point Crossover**
  - **N-Point Crossover**
  - **Uniform Crossover**

One-Point Crossover

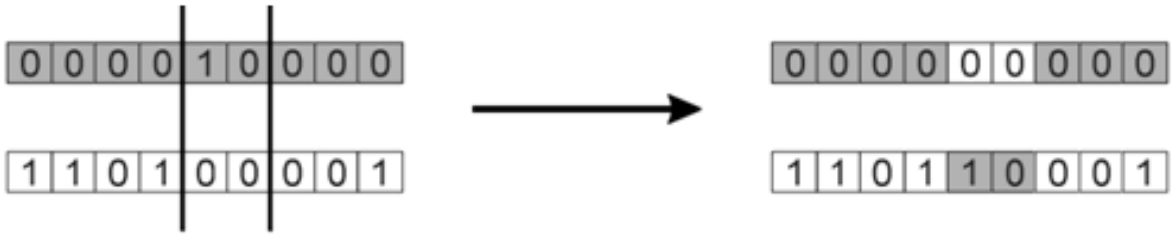
- Choose a random number in the range [0, *l* - *1*], with *l* the length of the encoding
- Split parents at this crossover point
- Create children by exchanging tails



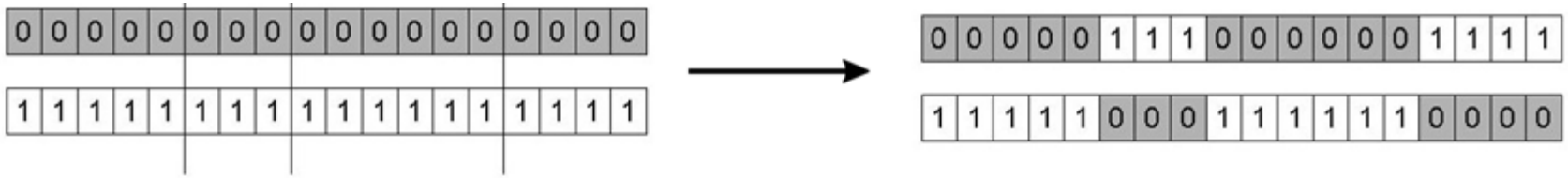
N-Point Crossover

- Generalisation of 1 point
- Choose n random crossover points
- Split along those points
- Join parts, alternating between parents

N-Point Crossover



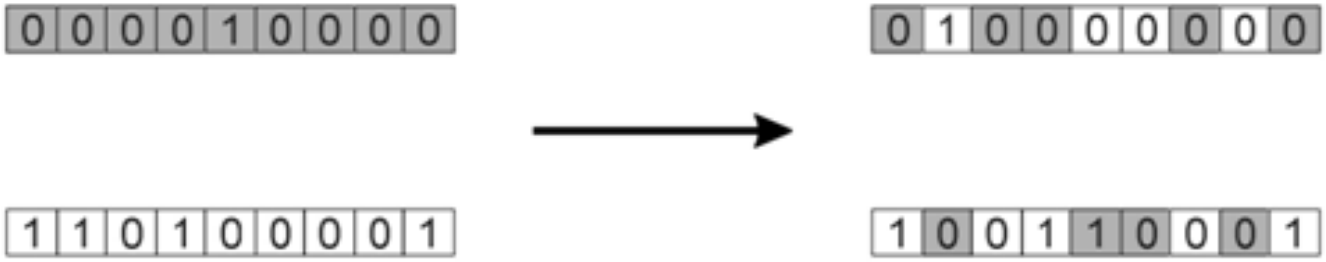
N=2



N=3

Uniform Crossover

- Uniform crossover works by treating each gene independently
- Making a random choice as to which parent it should be inherited from



Assume array: [0.35, 0.62, 0.18, 0.42, 0.83, 0.76, 0.39, 0.51, 0.36]

Recombination for Binary Representation

- **Positional bias**
  - e.g. in 1-point crossover bias against keeping bits at head and tail of string together
- **Distributional bias**
  - in uniform crossover bias is towards transmitting 50% of genes from each parent

Recombination for Integer Representation

- same as in binary representations
- N-point / uniform crossover operators work
- **Blending** is not useful
  - averaging even and odd integers produce a non-integer !

Recombination for Floating-Point Representation

- There two options for recombining two floating-point strings:
  - **Discrete Recombination**
  - **Intermediate or Arithmetic Recombination**

Recombination for Floating-Point Representation

- **Discrete Recombination:**
  - similar to crossover operators for bit-strings
  - alleles have floating-point representations
  - each allele value in offspring z comes from one of its parents (x, y) with equal probability: $z_i = x_i$ or $z_i = y_i$
  - Could use **n-point** or **uniform** crossover operators

Recombination for Floating-Point Representation

- **Intermediate or Arithmetic Recombination:**
  - for each gene position
  - new allele value between those of parents (x, y):
  - $z_i = \alpha\, x_i + (1 - \alpha)\, y_i$ where $\alpha : 0 \leq \alpha \leq 1$.
  - The parameter α can be:
    - constant: (**uniform arithmetical crossover)**, usually $\alpha = 0.5$
    - picked at random every time
    - variable (e.g. depend on the age of the population)

Recombination for Floating-Point Representation

- There three types of **arithmetic recombination**:
  - **Simple Arithmetic Recombination**
  - **Single Arithmetic Recombination**
  - **Whole Arithmetic Recombination**

Simple Arithmetic Recombination

- 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
- Put the first k floats of parent and put them into child
- The rest is arithmetic average of parent 1 and 2:

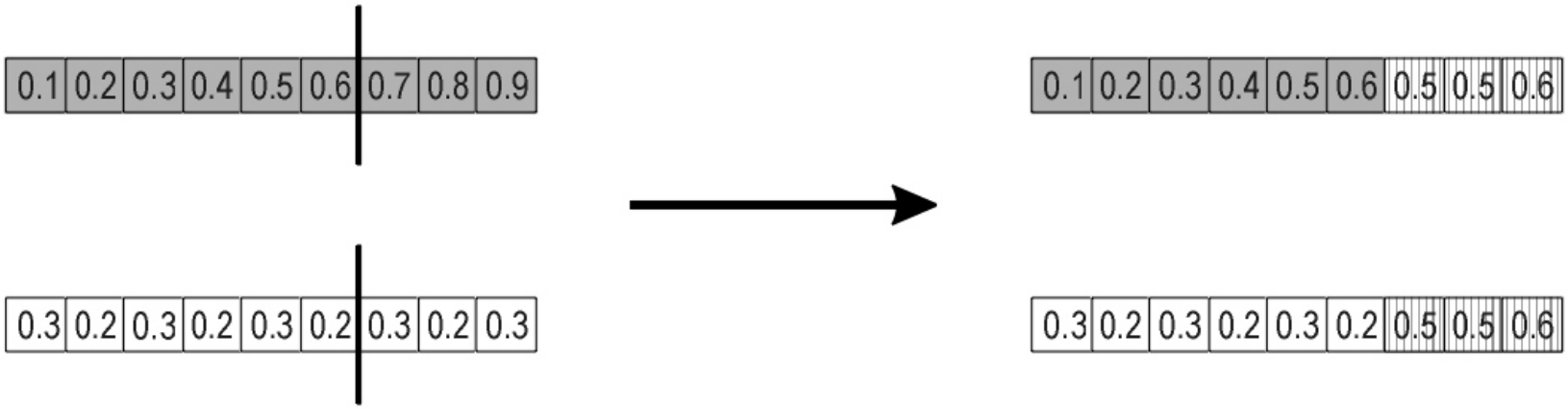
Child 1:

$$<x_1, \ldots, x_k, \ \alpha y_{k+1} + (1-\alpha) x_{k+1}, \ldots, \alpha y_n + (1-\alpha) x_n>$$

Child 2:

$$<y_1, \ldots, y_k, \ \alpha x_{k+1} + (1-\alpha) y_{k+1}, \ldots, \ \alpha x_n + (1-\alpha) y_n>$$

Simple Arithmetic Recombination



Example: k=6, α=0.5

Single Arithmetic Recombination

- Parents: $\langle x_1,\ldots,x_n \rangle$ and $\langle y_1,\ldots,y_n \rangle$
- Pick a single gene (*k*) at random
- At that position, take the arithmetic average of the two parents, the other points from the parents

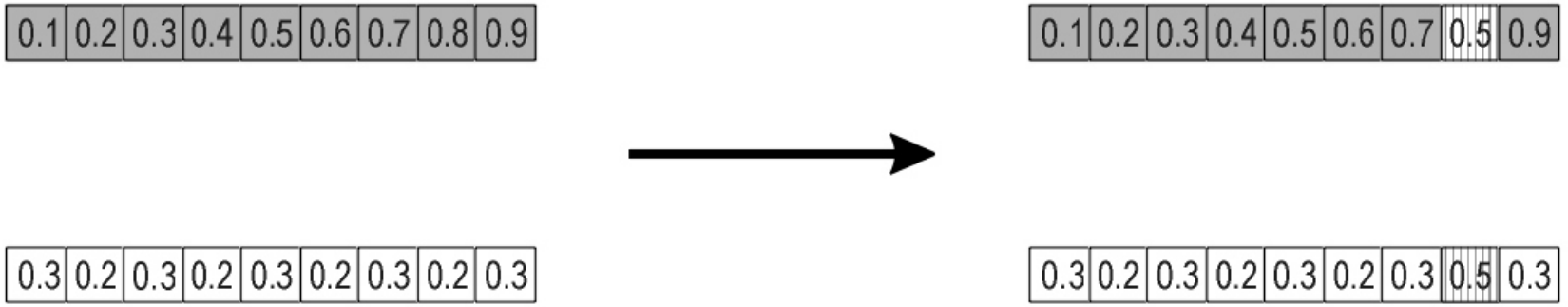
child1:

$$<x_1, \ldots, x_{k-1}, \ \alpha y_k + (1-\alpha) x_k, \ x_{k+1}, \ldots, \ x_n>$$

child2:

$$<y_1, \ldots, y_{k-1}, \ \alpha x_k + (1-\alpha) y_k, \ y_{k+1}, \ldots, \ y_n>$$

Single Arithmetic Recombination



Example: k=8, α=0.5

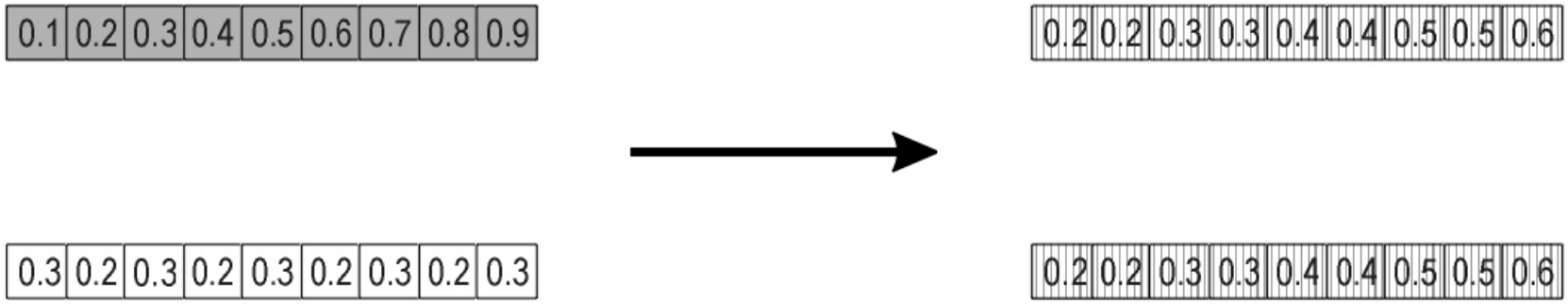
Whole Arithmetic Recombination

- Most commonly used
- Parents: $\langle x_1,\ldots,x_n \rangle$ and $\langle y_1,\ldots,y_n \rangle$
- takes weighted sum of the two parental alleles for each gene

$$Child1 = \alpha . x + (1-\alpha) . y$$

$$Child2 = \alpha . y + (1-\alpha) . x$$

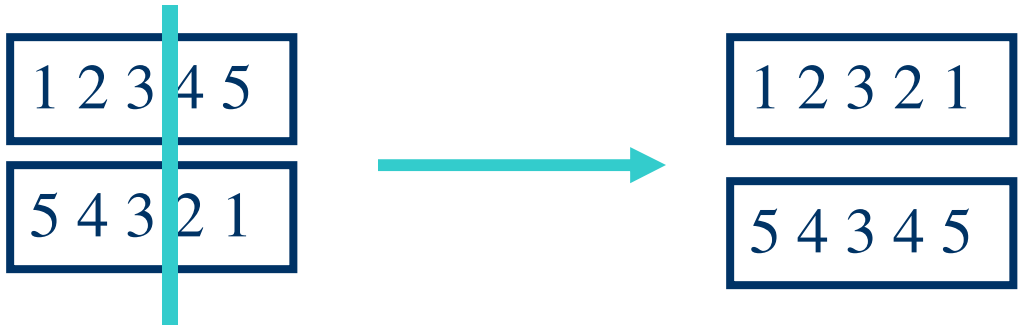
Whole Arithmetic Recombination



- **Note:** if α=0.5 two offspring are identical!

Recombination for Permutations Representation

- "Normal" crossover operators will often lead to inadmissible solutions



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

Recombination for Permutations Representation

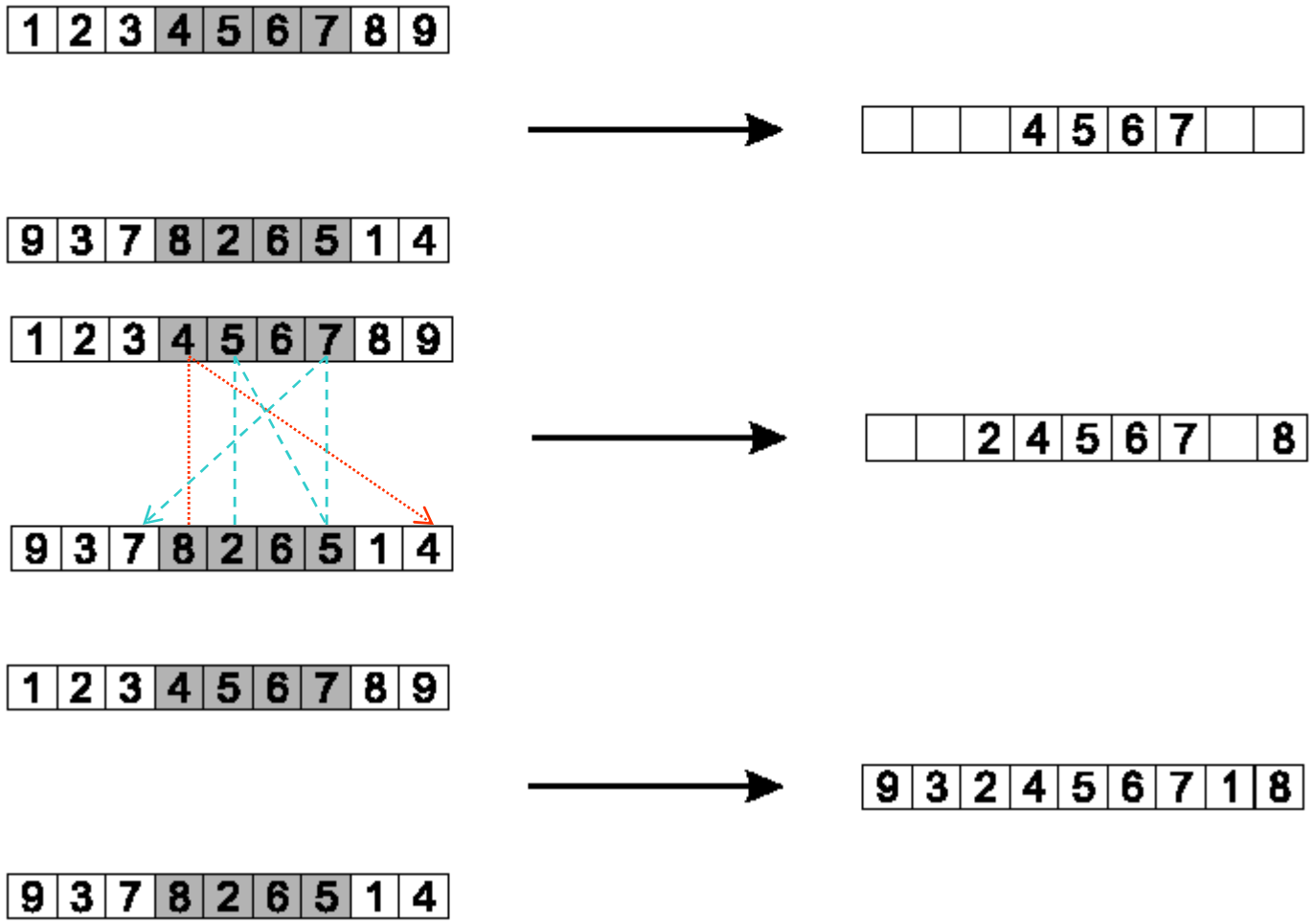
- Most commonly used operators:
  - For Adjacency-type Problems (e.g. TSP)
    - **Partially Mapped Crossover (PMX)**
    - **Edge Crossover**
  - For Order-type Problems (e.g. Job Shop Scheduling)
    - **Order Crossover**
    - **Cycle Crossover**

Partially Mapped Crossover (PMX)

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

Partially Mapped Crossover example



Edge Crossover

- Works by constructing a table listing which edges are present in the two parents, if an edge is common to both, mark with a +
- e.g. [1 2 3 4 5 6 7 8 9] and [9 3 7 8 2 6 5 1 4]

Element	Edges	Element	Edges
1	2,5,4,9	6	2,5+,7
2	1,3,6,8	7	3,6,8+
3	2,4,7,9	8	2,7+, 9
4	1,3,5,9	9	1,3,4,8
5	1,4,6+		

Edge Crossover

(once edge table is constructed)

1. Pick an initial element at random and put it in the offspring
2. Set the variable current element = entry
3. Remove all references to current element from the table
4. Examine list for current element:
   - If there is a **common edge**, pick that to be next element
   - Otherwise pick the entry in the list which itself has the shortest list
   - Ties are split at random
5. In the case of reaching an empty list:
   - Examine the other end of the offspring is for extension
   - Otherwise a new element is chosen at random

Edge Recombination example

Element	Edges	Element	Edges
1	2,5,4,9	6	2,5+,7
2	1,3,6,8	7	3,6,8+
3	2,4,7,9	8	2,7+, 9
4	1,3,5,9	9	1,3,4,8
5	1,4,6+		

Choices	Element selected	Reason	Partial result
All	1	Random	[*1*]
2,5,4,9	5	Shortest list	[*1 5*]
4,6	6	Common edge	[*1 5 6*]
2,7	2	Random choice (both have two items in list)	[*1 5 6 2*]
3,8	8	Shortest list	[*1 5 6 2 8*]
7,9	7	Common edge	[*1 5 6 2 8 7*]
3	3	Only item in list	[*1 5 6 2 8 7 3*]
4,9	9	Random choice	[*1 5 6 2 8 7 3 9*]
4	4	Last element	[*1 5 6 2 8 7 3 9 4*]

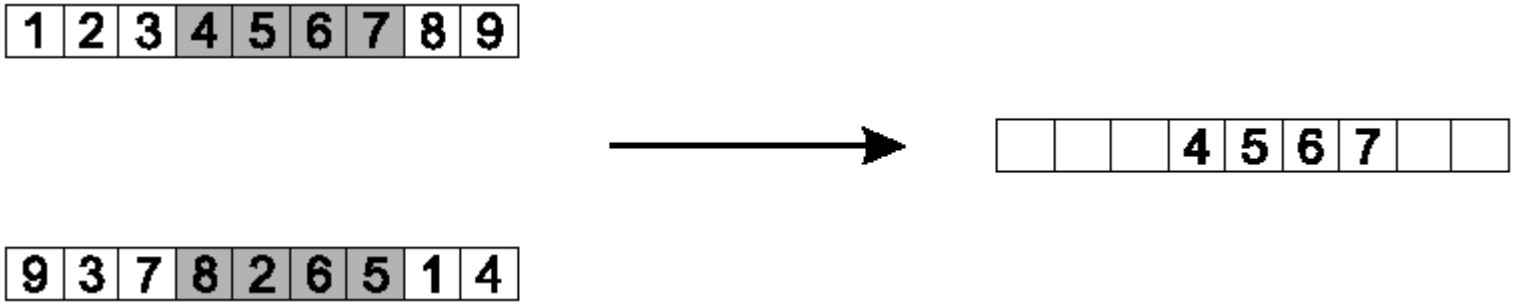
Parents: [1 2 3 4 5 6 7 8 9] and [9 3 7 8 2 6 5 1 4]

Order Crossover

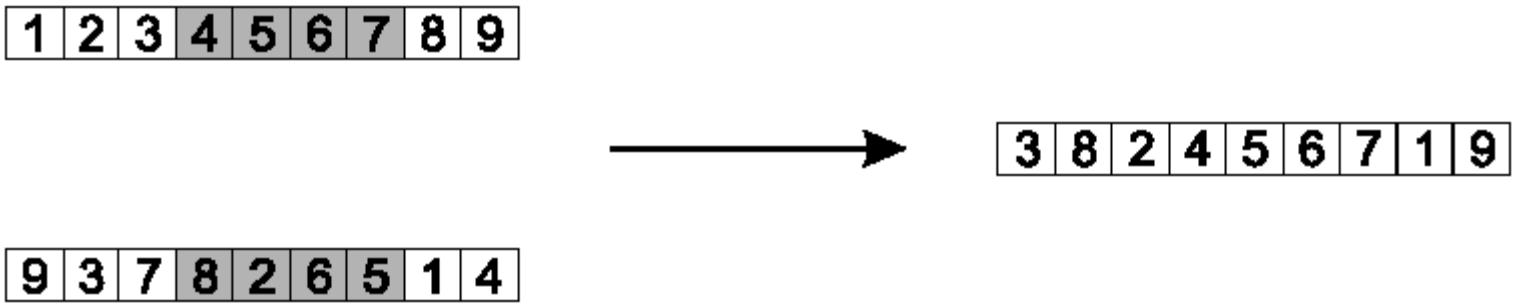
- Idea is to preserve relative order that elements occur
- Procedure:
  1. Choose an arbitrary part from the first parent
  2. Copy this part to the first child
  3. Copy the numbers that are not in the first part, to the first child:
     - starting right from cut point of the copied part,
     - using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

Order Crossover example

- Copy randomly selected set from first parent



- Copy rest from second parent in order 1,9,3,8,2



Cycle crossover

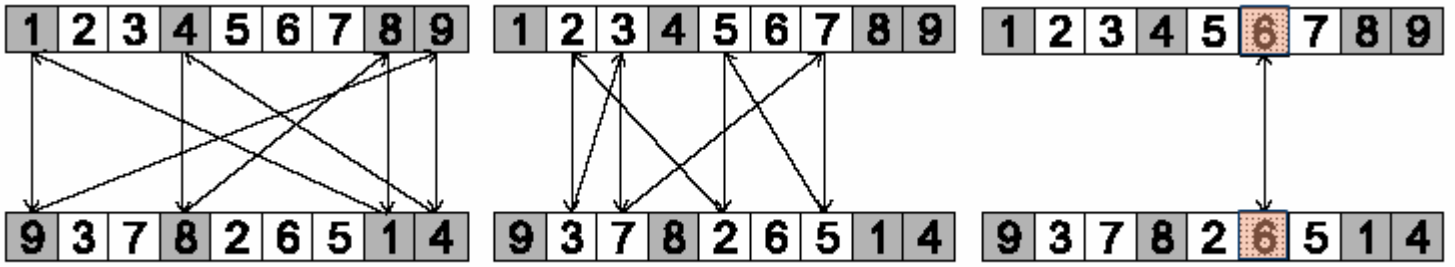
Basic idea: Each allele comes from one parent *together with its position*.

procedure:

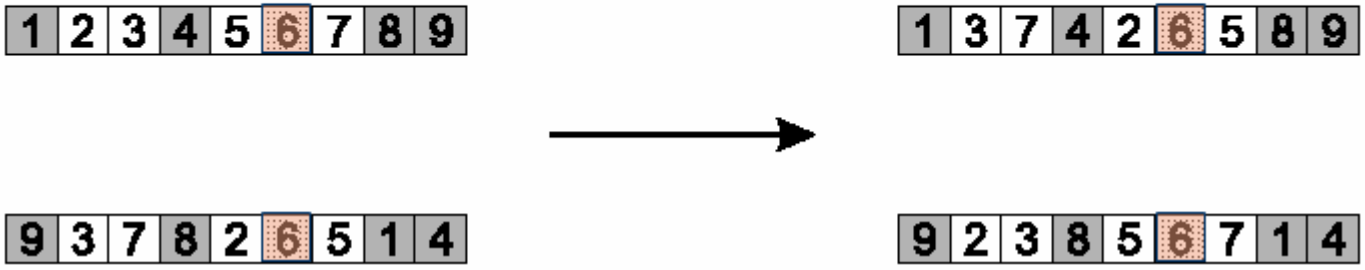
1. Make a cycle of alleles from P1 in the following way.
   (a) Start with the first allele of P1.
   (b) Look at the allele at the *same position* in P2.
   (c) Go to the position with the *same allele* in P1.
   (d) Add this allele to the cycle.
   (e) Repeat step b through d until you arrive at the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



Multi-Parent recombination

- Recall that we are not limited by the practicalities of nature
- Noting that mutation uses 1 parent, and "traditional" crossover 2, the extension to $a > 2$ is natural to examine
- Been around since 1960s, still rare but studies indicate useful

Multi-Parent recombination

- Three main types:
  - **Based on allele frequencies**
    - e.g., uniform crossover
  - **Based on segmentation and recombination of the parents**
    - e.g., n-point crossover
  - **Based on numerical operations on real-valued alleles**
    - e.g., generalising arithmetic recombination operators

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

Crossover OR mutation?

- **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 (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
  - e.g. thought experiment: 50% 0's on first bit in the population, 50% after performing n crossovers
- To hit the optimum you often need a 'lucky' mutation

Population Models

Population Models

- Population Models:
  - **Generational model**
  - **Steady state model**

Generational Model

- **Population of individuals** : size N
- **Mating pool** (parents) : size N
- **Offspring**
  - formed from parents
  - replace parents
  - are next generation : size N

Steady State Model

- Not whole population replaced
- N: population size (M < N)
  - M individuals replaced by M offspring
- **Generational Gap**
  - percentage of the population that is replaced
  - equal to M/N
- M=1 & generational gap of 1/N has widely applied

Parent Selection

Selection Types

- Selection can occur in two places:
  - Selection from current generation to take part in mating (**parent selection**)
  - Selection from parents + offspring to go into next generation (**survivor selection**)

Parent Selection

- The **parent selection** operation chooses an individual (chromosome) to be a parent for the next generation of the population, based on its fitness

Parent Selection

- **Selection scheme**: process that selects an individual to go into the mating pool

- **Selection pressure**: degree to which the better individuals are favoured
  - if higher selection pressure, better individuals favoured more
  - Determines convergence rate:
    - if too high, possible premature convergence
    - if too low, may take too long to find good solutions

Selection scheme

- **Selection scheme types:**
  - Fitness-Proportionate, e. g.:
    - **Roulette Wheel Selection (RWS)**
    - **Stochastic Universal Sampling (SUS)**
  - Ordinal based, e.g.:
    - **Ranking Selection**
    - **Tournament Selection**

Fitness-Proportionate Selection

- The probability that an individual f_i is selected for mating pool is

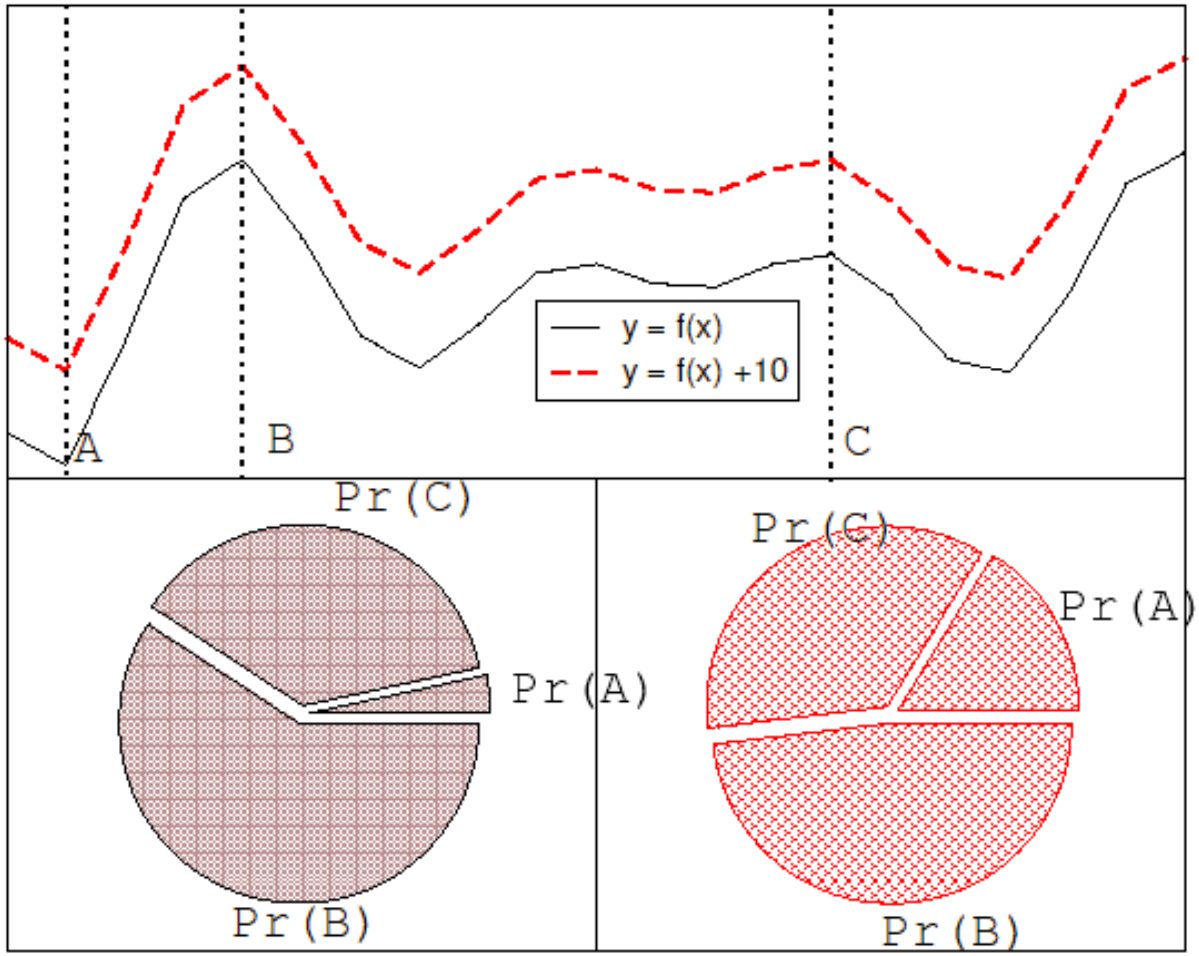
$$f_i \,/\, \sum_{j=1}^{\mu} f_j$$

- Selection probability depends on:
  - absolute **fitness of individual** compared to absolute fitness of **rest of the population**

Fitness Proportionate Selection

- Problems with FPS
  - **Premature Convergence**
    - One highly fit member can rapidly take over if rest of population is much less fit
    - Population converges to a local optimum
    - Too much exploitation; too few exploration
  - **Almost no selection pressure when fitness values close together**
  - **Fitness scaling effects**
    - May behave differently on transposed versions of same fitness function
    - e.g. consider f(x) and y(x)=f(x)+10;

Fitness Scaling Effects



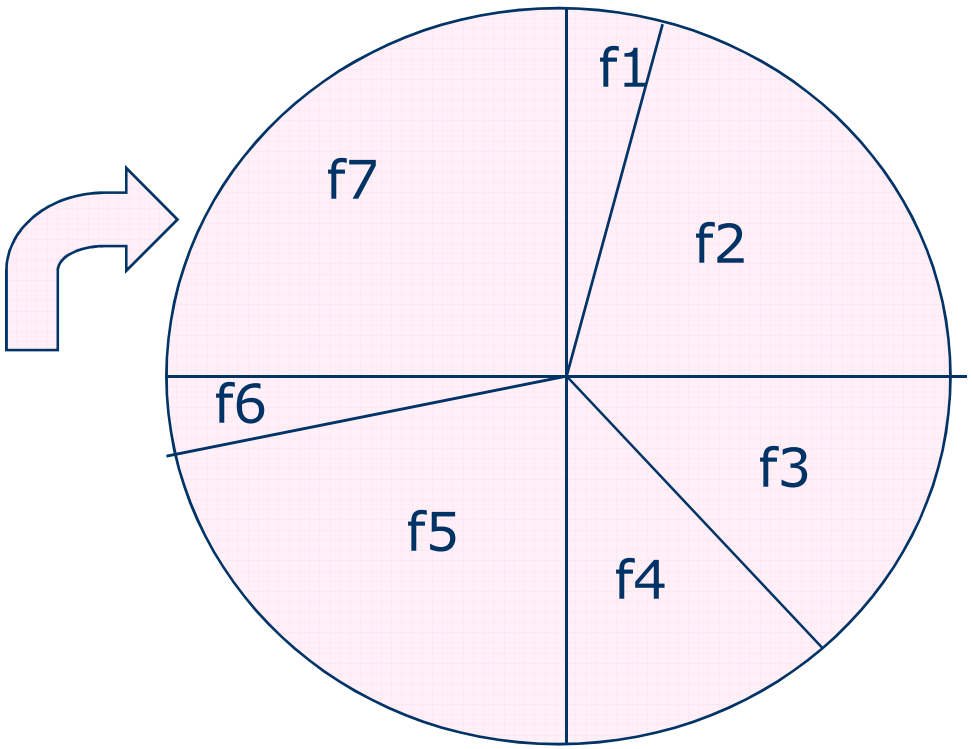
Fitness Proportionate Selection

- Scaling can fix last two problems
- **Windowing:**
  - $f'(i) = f(i) - \beta$
  - where β is worst fitness in this (last n) generations
- **Sigma Scaling:**
  - $f'(i) = \max(f(i) - (\mu_f - c \cdot \sigma_f), 0)$
  - where
    - μ_f is the mean of fitness values,
    - σ_f is the standard deviation of the fitness values, and
    - c is a constant, usually 2
  - if $f'(i) < 0$ then set $f'(i) = 0$

Roulette Wheel Selection

- **Main idea**: better individuals get higher chance
  - Chances are proportional to fitness
- Implementation
  - Assign to each individual a part of the roulette wheel
  - Spin the wheel n times to select n individuals

Roulette Wheel Selection



m=7

Roulette Wheel Algorithm

- $P_{sel}(i)$ is defined by the selection distribution:
  - fitness proportionate
  - ranking.

- $[a_1, a_2, \ldots, a_\mu]$ by: ($a_\mu = 1.0$)

$$a_i = \sum_1^i P_{sel}(i) \qquad i = 1,2,\ldots,m$$

  - m: population size

Roulette Wheel Algorithm

```
BEGIN
  set current_member = 1;
  WHILE ( current_member ≤ μ ) DO
    Pick a random value r uniformly from [0.1];
    set i = 1;
    WHILE ( a[i]< r ) DO
      set i = i + 1;
    OD
    set mating_pool[current_member] = parents[i];
    set current_member = current_member + 1;
  OD
END
```

Roulette Wheel Algorithm

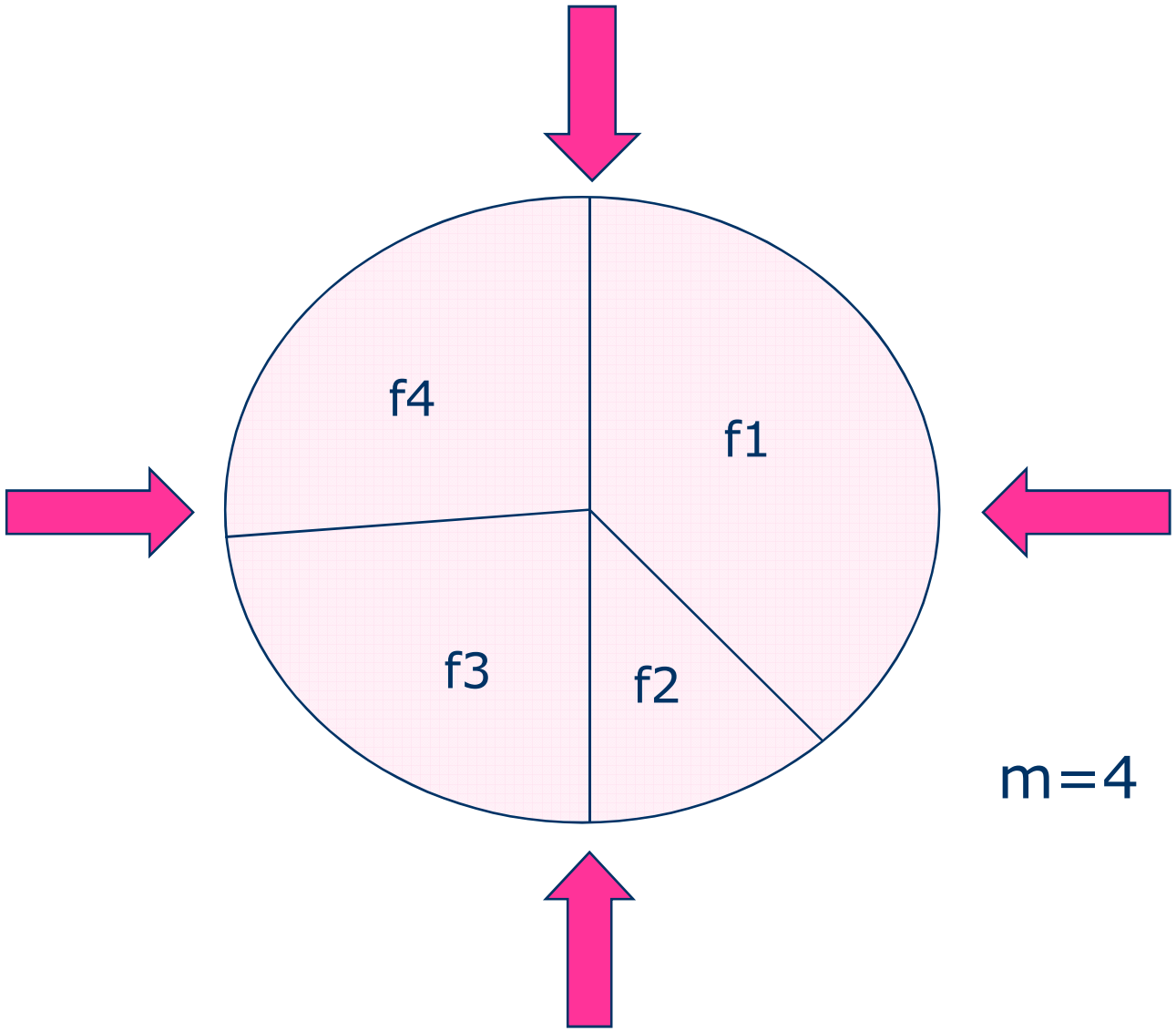
Fitness based weighting: assign distributions by fitness

Chromosome	Fitness	p_i	a_i
00110010001100	13477	0.265	0.265
11101100000001	12588	0.248	0.513
00101111001000	12363	0.243	0.756
00101111000110	12359	0.243	1.0

Stochastic Universal Sampling

- **Stochastic Universal Sampling (SUS)** spins the wheel once—but with *M* equally spaced pointers, which are used to selected the *M* parents.

Stochastic Universal Sampling



Stochastic Universal Sampling

```
BEGIN
  set current_member = i = 1;
  Pick a random value r uniformly from [0. 1/μ];
  WHILE ( current_member ≤ μ ) DO
    WHILE (  r ≤ a[i] ) DO
      set mating_pool[current_member] = parents[i];
      set r = r + 1/μ;
      set current_member = current_member + 1;
    OD
    set i = i + 1;
  OD
END
```

Ranking Selection

- Ordinal based method
- Attempt to remove problems of FPS by basing selection probabilities on **relative** rather than **absolute** fitness
- Population sorted by fitness
- Selection probabilities based on rank, not to their actual fitness
- The actual fitness value is less important, it is the rank in this order what matters
- Constant selection pressure

Ranking Selection

- This imposes a **sorting** overhead on the algorithm, but this is usually negligible compared to the fitness evaluation time

- How to allocate probabilities to ranks
  - can be any linear or non-linear function
  - e.g. **linear ranking selection (LRS)**

Linear Ranking

$$P_{lin-rank}(i) = \frac{(2-s)}{\mu} + \frac{2i(s-1)}{\mu(\mu-1)}$$

- Parameterised by factor *s:* $1.0 < s \leq 2.0$
  - measures advantage of best individual
  - in generational GA `s`: no. of expected offspring allotted to best
- Assume best has rank μ and worst *1*

Linear Ranking

- Simple 3 member example

	Fitness	Rank	P_{selFP}	$P_{selLR} \quad (s = 2)$	$P_{selLR} \quad (s = 1.5)$
A	1	1	0.1	0	0.167
B	5	3	0.5	0.67	0.5
C	4	2	0.4	0.33	0.33
Sum	10		1.0	1.0	1.0

Tournament Selection

- Ordinal based
- RWS and SUS uses info on whole population
  - info may not be available
    - population too large
    - population distributed on a parallel system

Tournament Selection

- Relies on an ordering relation to rank any n individuals
- Most widely used approach
- Tournament size k
  - if k large, more of the fitter individuals
  - controls selection pressure
    - k=2 : lowest selection pressure
    - higher *k* increases selection pressure

Tournament Selection

```
BEGIN
  set current_member = 1;
  WHILE ( current_member ≤ μ ) DO
    Pick k individuals randomly, with or without replacement;
    Select the best of these k comparing their fitness values;
    Denote this individual as i;
    set mating_pool[current_member] = i;
    set current_member = current_member + 1;
  OD
END
```

Assumes that a tournament is held at this point

μ: population size k: tournament size

Survivor Selection

Survivor Selection

- Also known as replacement
- Determines who survives into next generation
  - reduces (m+l) to m
    - m population size (also no. of parents)
    - l no. of offspring at end of generation
- several replacement strategies

Survivor Selection

- Survivor selection can be divided into two approaches:
  - Age-Based Selection
    - FIFO
    - Replace random
  - Fitness-Based Selection
    - Elitism
    - GENITOR

Age-Based Replacement

- Fitness not taken into account
- Each inidividual exists for same number of generations
  - in SGA only for 1 generation
- e.g. create 1 offspring and insert into population at each generation
  - FIFO
  - Replace random (**not recommended**)

Fitness-Based Replacement

- Elitism
  - Always keep the best or the best few of the of the fittest solution so far
  - Ensures that the best found solution (s) are never lost
- GENITOR: a.k.a. “delete-worst”
  - Rapid takeover: use with large populations or “no duplicates” policy
  - fast increase in population mean
  - possible premature convergence

Glossary

Glossary

- **allele**: a variant of a gene, i.e. the value of a symbol in a specified position of the genotype.
- **chromosome**: synonymous to “genotype”.
- **crossover**: combination of two individuals to form one or two new individuals.
- **fitness function**: function giving the value of an individual.
- **generation**: iteration of the basic loop of an genetic algorithm.

Glossary

- **gene**: an element of a genotype, i.e. one of the symbols of a symbol string.
- **genotype**: a symbol string generating a phenotype at the time of a decoding phase.
- **individual**: an instance of solution for a problem dealt with by genetic algorithm.
- **locus**: position of a gene in the genotype.
- **mutation**: random modification of an individual.
- **search operator**: synonymous to “variation operator”.

Glossary

- **replacement operator**: determines which individuals of a population will be replaced by the offspring. It thus makes it possible to create the new population for the next generation.
- **selection operator**: determines how much time a “parent” individual generates “offspring” individuals.
- **variation operator**: operator modifying the structure, the parameters of an individual, such as the crossover and the mutation.

Glossary

- **phenotype**: set of the observable appearances of the genotype. More specifically, it is an instance of solution for the problem dealt with, expressed in its natural representation obtained after decoding the genotype.
- **population**: the set of the individuals who evolve simultaneously under the action of an evolutionary algorithm.
- **recombination**: synonymous to “crossover”.

References

References

- Eiben and Smith. **Introduction to Evolutionary Computing**, Springer-Verlag, New York, 2003.
- J. Dreo A. Petrowski, P. Siarry E. Taillard, **Metaheuristics for Hard Optimization**, Springer-Verlag, 2006.

The End