Genetic Algorithms (GA)

The most widely known type of Evolutionary Algorithms

Chapter 3

GA Quick Overview

- Developed: USA (Ann Arbor, Michigan) in the 1970's
- Early names: J. Holland (1962, 1975), K. DeJong, D. Goldberg
- Typically applied to:
  - discrete optimization
- Attributed features:
  - not too fast
  - good heuristic for combinatorial problems
- Special Features:
  - Traditionally emphasizes combining information from good parents (crossover)
  - many variants, e.g., representation forms (binary, integer, real-valued (floating point), permutation), reproduction models, operators

GA Quick Overview (2)

– A genetic algorithm can be considered an **iterative** search procedure / global optimization. ➔ not too fast

– The algorithm processes a population of potential solutions (chromosomes) distributed over the entire search space.

- John Holland (Schemata Theorem - 1970)
- 1975 J. Holland published **Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence**
  - Considered by most to be the seminal work in the field
  - Established formal, theoretical basis for evolutionary optimization with introduction of schemata (building blocks, partial solutions)
- Keneth DeJong

  1975 Kenneth De Jong's thesis, under Holland, **introduced a set of test functions with distinct properties, demonstrating the broad applicability of Gas**
- David Goldberg (1989 - *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley)

  1983 - *Computer-aided gas pipeline operation using genetic algorithms and rule learning*, PhD thesis (under Holland), University of Michigan. Ann Arbor, MI.

GA Quick Overview (3)

- First formulated by Holland for adaptive search and by his students for optimizations from mid 1960s to mid 1970s.
- Binary strings have been used extensively as individuals (chromosomes).
- Simulate Darwinian evolution.
- Search operators are only applied to the *genotypic* representation (chromosome) of individuals.
- Emphasize the role of recombination (crossover). Mutation is only used as a background operator.
- Often use roulette-wheel (Monte Carlo) selection.

Genetic algorithms

- Holland's original GA is now known as the simple genetic algorithm (SGA)
- Other GAs use different:
  - Representations
  - Mutations
  - Crossovers
  - Selection mechanisms

Genetic Algorithms (pseudocod I)

- An Example of a Genetic Algorithm:

```
Procedure GA{
  t = 0;
  Initialize P(t);
  Evaluate P(t);
  While (Not Done)
  {
    Parents(t) = Select_Parents(P(t));
    Offspring(t) = Procreate(Parents(t));
    Evaluate(Offspring(t));
    P(t+1)= Select_Survivors(P(t),Offspring(t));
    t = t + 1;
  }
```

Genetic Algorithms (pseudocod II)

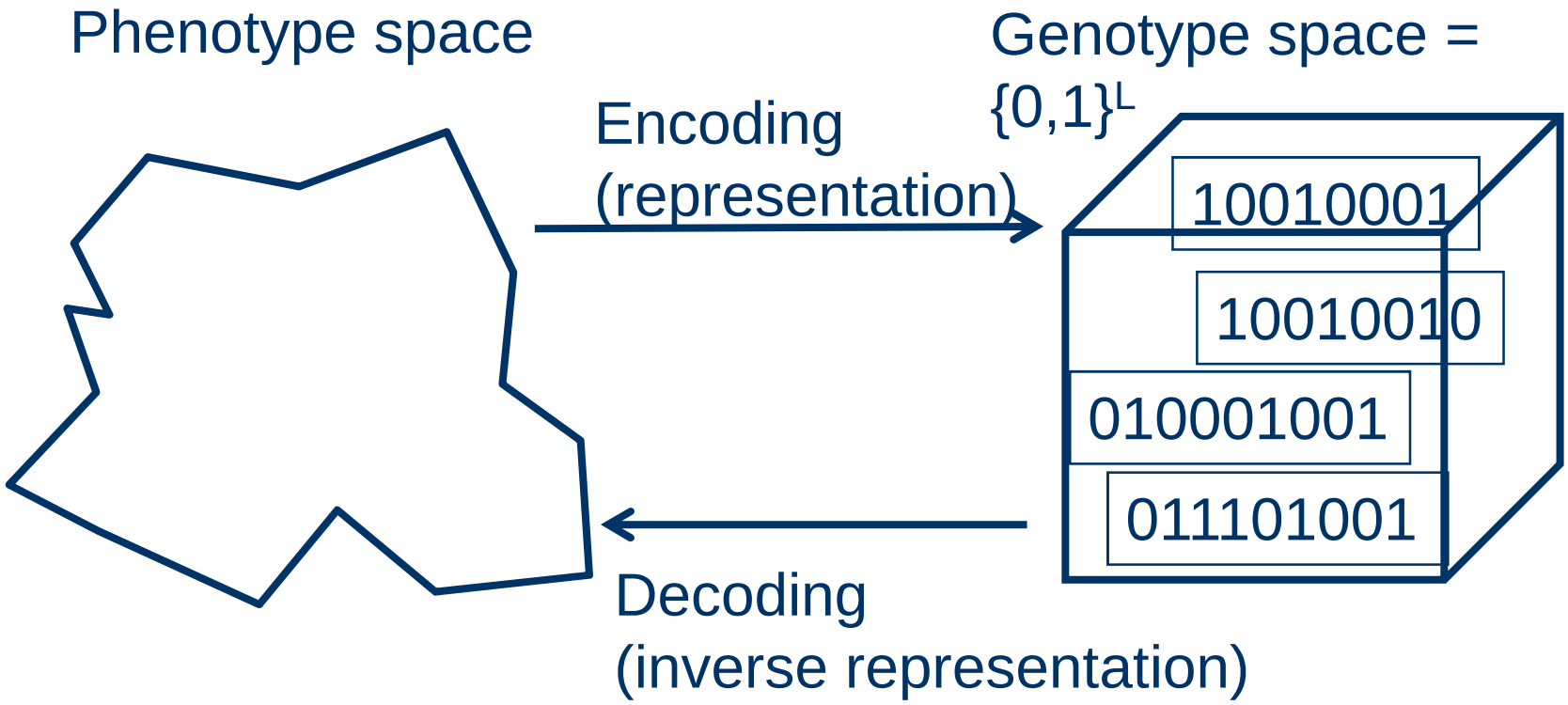
- P1: Choose a selection mechanism and a mechanism for scaling, if necessary.
- P2: Set `t = 0;` and **`Initialize`** `the population P(t).`
- P3: It **evaluates** the performance of each chromosome from `P(t)`. It retains most powerful individual in the population `P(t).`
- P4: **Selection operator** is applied *n* times *(n* – number of individuals*).* The selected chromosomes form an **intermediate population $\mathbf{P_1}$** (having also *n* members). In $\mathbf{P_1}$, some chromosomes of `P(t)` can have more children (will appear more than once) and others have no copy.
- P5: Apply **crossover operator** on the chromosomes of intermediate population $\mathbf{P_1}$. The chromosomes obtained form a new population $\mathbf{P_2}$. Deleted from the intermediate population $\mathbf{P_1}$, the parents of chromosomes obtained by crossover. The remaining chromosomes from $\mathbf{P_1}$ become members of the $\mathbf{P_2}$ population.
- P6: Apply **mutation operator** on the chromosomes of population $\mathbf{P_2}$. The result is the **new generation** `P(t+1)`.
- P7: Set `t = t + 1; If t≤N,` where `N` is the maximum number of generation, then continues with step `P3. Otherwise STOP.`

SGA technical summary tableau

Representation	Binary strings
Recombination	N-point or uniform
Mutation	Bitwise bit-flipping with fixed probability
Parent selection	Fitness-Proportionate
Survivor selection	All children replace parents
Speciality	Emphasis on crossover

Representation

Representation - The **most critical decision** in any application, namely that of **deciding how best to represent a candidate solution of the algorithm**.



SGA reproduction cycle

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

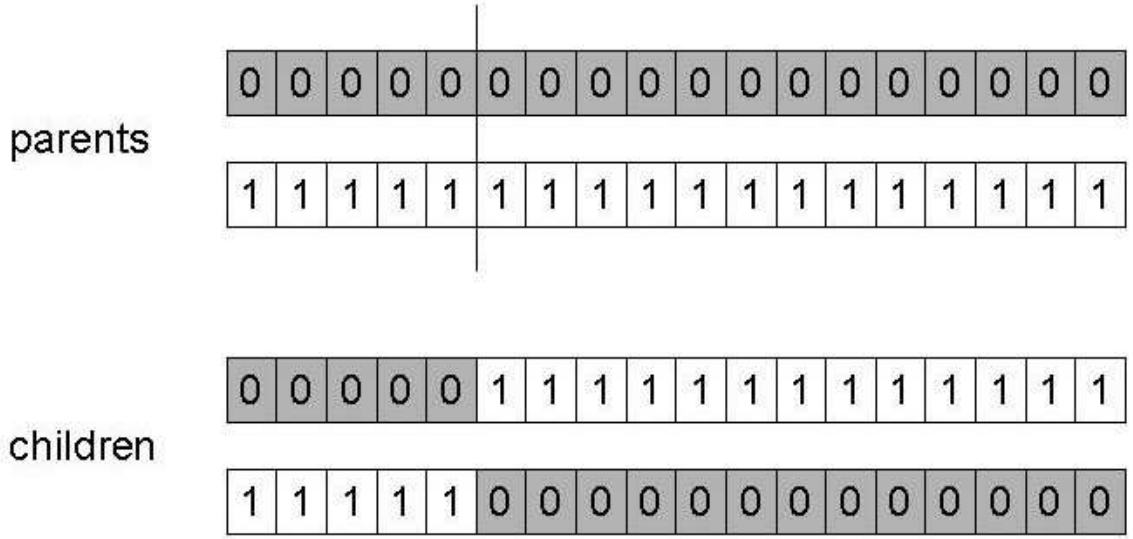
SGA operators: recombination / crossover

1. One-point crossover
2. k-point crossover (*k > 1), uniform vs. non-uniform*
3. Uniform crossover

Crossover rate: The probability of applying crossover.

SGA operators: 1-point crossover

- Choose a random point on the two parents
- Split parents at this crossover point
- Create children by exchanging tails
- P_c typically in range (0.6, 0.9)



SGA operators: mutation (1)

1. Bit-flipping
2. Random bit assignment
3. Multiple bit mutations

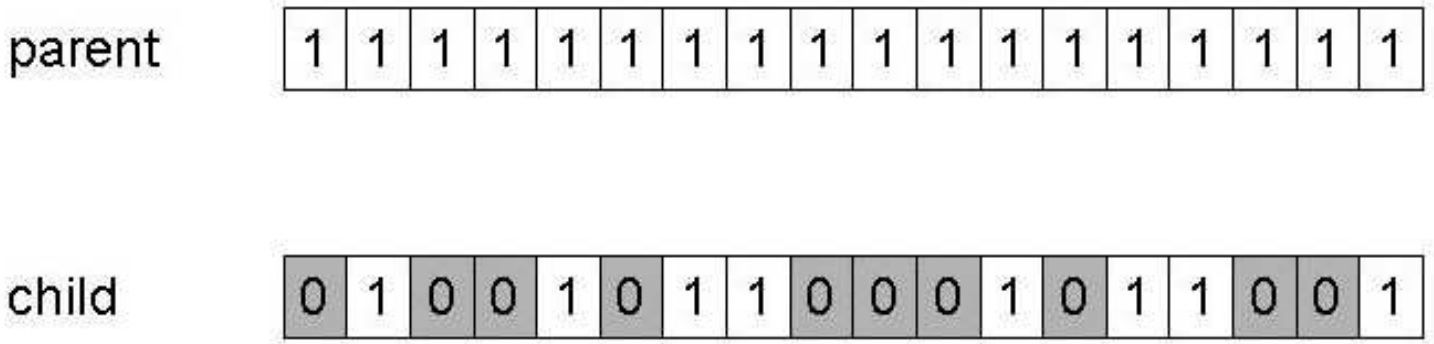
Each gene of chromosomes in the population may suffer a mutation. In a chromosome there may be several positions undergoing mutation.

Mutation rate:

Note the *difference between per bit (gene) and per* chromosome (individual) mutation rates.

SGA operators: mutation (2)

- 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
  - Order aprox. 10^{-3}



Search Bias

When a search operator is used, some offspring tend to be more likely to be generated than others. We called such a tendancy as *bias*.

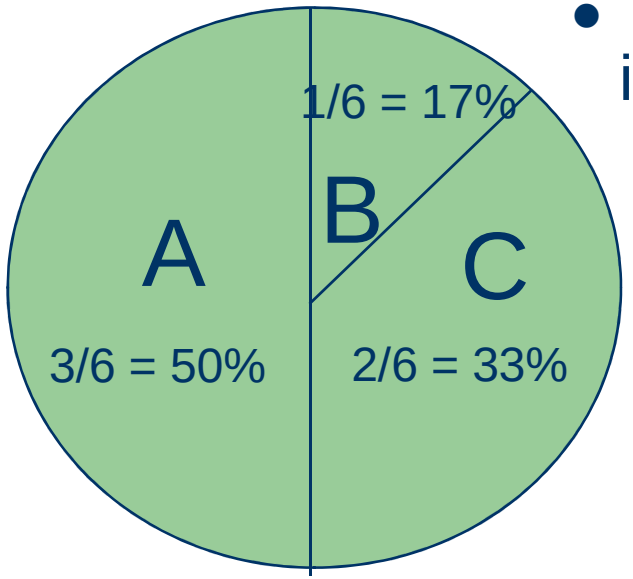
- *Crossover bias*, e.g., 1-point crossover vs. uniform crossover

  For crossover operators which exchange contiguous sections of the chromosomes (e.g. k-point) **the ordering of the variables may become important**. This is particularly true when good solutions contain *building blocks* which might be disrupted by a non-respectful crossover operator.

- *Mutation bias*, e.g., flipping 1 bit vs. flipping *k* bits

SGA operators: Selection

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



fitness(A) = 3

fitness(B) = 1

fitness(C) = 2

Genetic Algorithms: Proportionate Selection

- In Proportionate Selection, individuals are assigned a probability of being selected based on their fitness:
  - $p_i = f_i / \Sigma f_j$
  - Where p_i is the probability that individual i will be selected,
  - f_i is the fitness of individual i, and
  - Σf_j represents the sum of all the fitnesses of the individuals with the population.
- This type of selection is similar to using a *roulette wheel* where the fitness of an individual is represented as proportionate slice of wheel. The wheel is then spun and the slice underneath the wheel when it stops determine which individual becomes a parent.

Genetic Algorithms: Roulette wheel selection scheme

1 Evaluate the fitness, f_i, of each individual in the population.

2 Compute the probability (slot size), p_i, of selecting each member of the population: $p_i = f_i / \sum_{j=1}^{n} f_j$, where n is the population size.

3 Calculate the cumulative probability, q_i, for each individual: $q_i = \sum_{j=1}^{i} p_j$.

4 Generate a uniform random number, $r \in (0, 1]$.

5 If $r < q_1$ then select the first chromosome, x_1, else select the individual x_i such that $q_{i-1} < r \leq q_i$.

6 Repeat steps 4–5 n times to create n candidates in the mating pool.

Genetic Algorithms: Proportionate Selection

- There are a number of disadvantages associated with using proportionate selection:
  - Cannot be used on minimization problems,
  - Domination of *Super Individuals* in early generations and slow convergence in later generations.
  - Loss of selection pressure (search direction) as population converges.
- *Fitness scaling* (*linear, exponential, etc*) has often been used in early days to combat the two problems.

Genetic Algorithms: *Fitness Scaling* (I)

- **Simple scaling**:
  - The *i*th individual's fitness is defined as*:*

$$f_{scaled}(t) = f_{original}(t) - f_{worst}(t),$$

where *t* is the *generation number* and $f_{worst}(t)$ the *fitness of* the worst individual so far.

- **Sigma scaling**:
  - The *i*th individual's fitness is defined as*:*

$$f_{scaled}(t) = \begin{cases} f_{original}(t) - (\bar{f}(t) - c\sigma_f(t)), & \text{if } f_{scaled}(t) > 0 \\ 0, & \text{otherwise} \end{cases}$$

where *c* is a constant*, e.g., 2,* $\bar{f}(t)$ is the *average fitness* in the current population, and $\sigma_f(t)$ is the *standard deviation* of the fitness in the current population.

Genetic Algorithms: *Fitness Scaling* (II)

- **Power scaling**:
  - The *i*th individual's fitness is defined as*:*

$$f_{scaled}(t) = (f_{original}(t))^k,$$

where $k > 0$.

- **Exponential scaling**:
  - The ith individual's fitness is defined as:

$$f_{scaled}(t) = exp(f_{original}(t)/T)$$

where $T > 0$ is the *temperature*, approaching zero.

An example after Goldberg ‘89 (1)

- Simple problem: max x^2 over {0,1,…,31}
- GA approach:
  - Representation: binary code, e.g. 01101 $\leftrightarrow$ 13
  - Population size: 4
  - 1-point xover, bitwise mutation
  - Roulette wheel selection
  - Random initialisation
- We show one generational cycle done by hand

x^2 example: selection

String no.	Initial population	x Value	Fitness $f(x) = x^2$	$Prob_i$	Expected count	Actual count
1	0 1 1 0 1	13	169	0.14	0.58	1
2	1 1 0 0 0	24	576	0.49	1.97	2
3	0 1 0 0 0	8	64	0.06	0.22	0
4	1 0 0 1 1	19	361	0.31	1.23	1
Sum			1170	1.00	4.00	4
Average			293	0.25	1.00	1
Max			576	0.49	1.97	2

$$p_i = \frac{f_i}{\sum_j f_j}$$

X^2 example: crossover

String no.	Mating pool	Crossover point	Offspring after xover	x Value	Fitness $f(x) = x^2$
1	0 1 1 0 | 1	4	0 1 1 0 0	12	144
2	1 1 0 0 | 0	4	1 1 0 0 1	25	625
2	1 1 | 0 0 0	2	1 1 0 1 1	27	729
4	1 0 | 0 1 1	2	1 0 0 0 0	16	256
Sum					1754
Average					439
Max					729

X^2 example: mutation

String no.	Offspring after xover	Offspring after mutation	x Value	Fitness $f(x) = x^2$
1	0 1 1 0 0	1 1 1 0 0	26	676
2	1 1 0 0 1	1 1 0 0 1	25	625
2	1 1 0 1 1	1 1 0 1 1	27	729
4	1 0 0 0 0	1 0 1 0 0	18	324
Sum				2354
Average				588.5
Max				729

The simple GA

- Has been subject of many (early) studies
  - still often used as benchmark for novel GAs
- Shows many shortcomings, e.g.
  - Representation is too restrictive
  - Mutation & crossovers only applicable for bit-string & integer representations
  - Selection mechanism sensitive for converging populations with close fitness values
  - Generational population model (step 5 in SGA repr. cycle) can be improved with explicit survivor selection

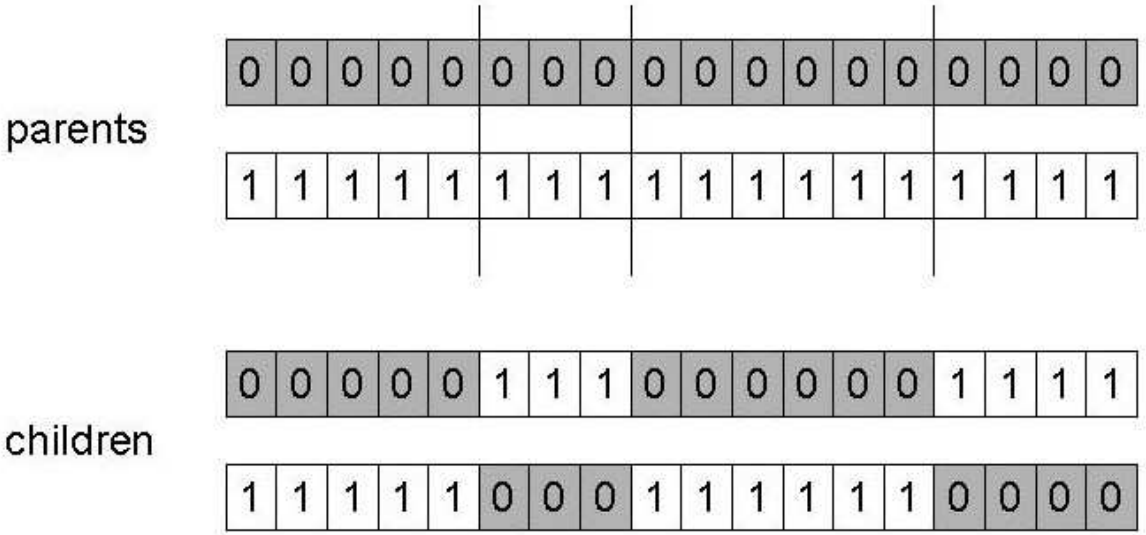
Alternative Crossover Operators

- Performance with 1 Point Crossover depends on the order that variables occur in the representation
  - more likely to keep together genes that are near each other
  - Can never keep together genes from opposite ends of string
  - This is known as *Positional Bias*
  - Can be exploited if we know about the structure of our problem, but this is not usually the case
- Disadvantage: using a 1-point crossover some combinations of genes cannot be obtained.
- Let consider the following example: we have two performing scheme – S1 = (0 1 * * * * * * * 1 1)

  – S2 = (* * * 1 0 1 * * * * *). Combining the two chromosomes that represent these schemes will never lead to the scheme – S3 = (0 1 * 1 0 1 * * * 1 1).

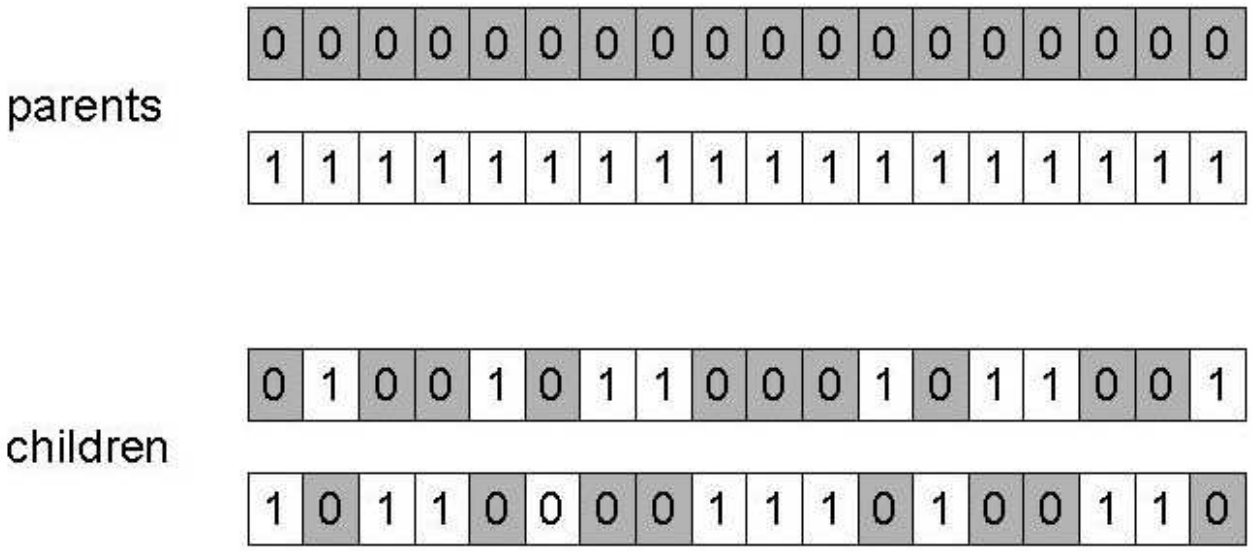
n-point crossover

- Choose n random crossover points
- Split along those points
- Glue parts, alternating between parents
- Generalisation of 1 point (still some positional bias)



Uniform crossover

- Do not use cut points
- Assign 'heads' to one parent, 'tails' to the other
- Flip a coin for each gene of the first child
- Make an inverse copy of the gene for the second child
- Inheritance is independent of position



For each gene of a descendant, a global parameter indicates the probability that this gene come from the first (or second) parent.

Hux (half uniform crossover)

- **In this operator, bits are randomly and independently exchanged, but exactly half of the bits that differ between parents are swapped.**
- **The HUX operator ensures that the offspring are equidistant between the two parents. This serves as a diversity preserving mechanism.**
- The two parent individuals are combined to produce two new offspring. Exactly half of the non-matching bits in the parent individuals are swapped. This gives the number of differing bits which is divided by two. The resulting number is how many of the bits that do not match between the two parents are swapped.

Example

Parent A	1	1	0	1	1	0	1	1
Parent B	0	0	1	1	1	0	1	0
* Different Alleles	*	*	*	1	1	0	1	*
x Allels to Interchange	x	*	x	1	1	0	1	*
Offspring A	0	1	1	1	1	0	1	1
Offspring B	1	0	0	1	1	0	1	0

Alternative Mutation Operators

Through mutation are introduced in the population individuals which could not be generated by other mechanisms.

- Strong Mutation
- Weak Mutation
- Single chromosome (individual) mutation
- Not uniform mutation
- Adaptive not-uniform mutation

Strong Mutation

In the strong form of the operator, the position selected for mutation is automatically changed.

Algorithm:

P1: For each chromosome of the current population and for each position of the chromosome is performed:

P1.1. It generates a random number q in the interval [0,1].

P1.2. If $q < p_m$ then perform mutation of that position, changing 0 to 1 and 1 to 0. Otherwise (if $q \geq p_m$), the position remains unchanged.

Weak Mutation

In the weak form of the operator, the position selected for mutation is NOT automatically changed.

New value of the position which meets the condition of mutation, is generated randomly.

Algorithm:

P1: For each chromosome of the current population and for each position of the chromosome is performed:

P1.1. It generates a random number *q* in the interval [0,1].

P1.2'. If $q < p_m$ then randomly choose one of the values 0 or 1. Current position is assigned as the value selected. Otherwise (if $q \geq p_m$), the position remains unchanged.

Single chromosome (individual) Mutation

- In this version, each application of the operator affects a gene (or possibly more) of a chromosome (which was selected for mutation).

- You can use both the weak and the strong version of the mutation operator.

Not uniform mutation

- It can be considered a mutation probability which **depends on the generation *(not uniform).***
  - In the first generation the probability of mutation is high and decreases with the index ***t*** of her generation.
  - In this way ensure big changes (**quick progress**) in the ***early stages*** *of search*.
  - Decrease the probability of mutation in the ***advanced stages*** *of search* generates small changes, which ensure a more **efficient local search**.
- $p_m(t) = p_m * e^{(1-t)\beta}$

  where p_m is the mutation probability at first generation, $\beta \geq 1$ (real parameter); β ↗ (as quick increases) ➔ $p_m(t)$ ↘ (as quick decreases)

Adaptive not-uniform mutation

- The mutation probability depends on the position (gene rank) in chromosome and generation (***t***).
  - In the **first generations** perform a **global search**
  - In the **last part of the process** it performs a **local search**
  - As the generation index (***t***) increases the probability of mutation of the first genes from chromosome to decrease, and of the last genes to grow.
- Example:
  - Let's consider the search space being $\Re$. The chromosomes binary encode the real values. **If a gene is positioned at the beginning of a chromosome, then its mutation it causes a significant change of the chromosome.** Elements of search space corresponding to the two chromosomes (at the beginning and that containing gene changed) will be very different.
  - **If mutation occurs at the end of the chromosome will result in a much smaller change.**

Crossover OR mutation?

- Decade long debate: which one is better / necessary / main-background
- Answer (at least, rather wide agreement):
  - it depends on the problem, but
  - in general, it is good to have both
  - both have another role
  - mutation-only-EA is possible, xover-only-EA would not work
- Achieving a balance between information **exploitation** and by the state-space **exploration** to obtain new better solutions, is typical of all powerful optimization methods.
- If the solutions obtained are exploited too much, then reaches a **premature convergence**.
- On the other hand, if too much emphasis on exploration, it is possible that the **information already obtained is not used properly**. Search time grows and approaches that of random search methods.

Crossover OR mutation? (cont'd)

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

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

There is co-operation AND competition between them

- Crossover is explorative, it makes a *big* jump to an area somewhere "in between" two (parent) areas
- Mutation is exploitative, it creates random *small* diversions, thereby staying near (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

Other representations

- Gray coding of integers (still binary chromosomes)
  - Gray coding is a mapping that means that small changes in the genotype cause small changes in the phenotype (unlike binary coding). “Smoother” genotype-phenotype mapping makes life easier for the GA

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. image processing parameters
- Others take *categorical* values from a fixed set e.g. {blue, green, yellow, pink}
- N-point / uniform crossover operators work
- Extend bit-flipping mutation to make
  - “creep” i.e. more likely to move to similar value
  - Random choice (esp. categorical variables)
  - For ordinal problems, it is hard to know correct range for creep, so often use two mutation operators in tandem

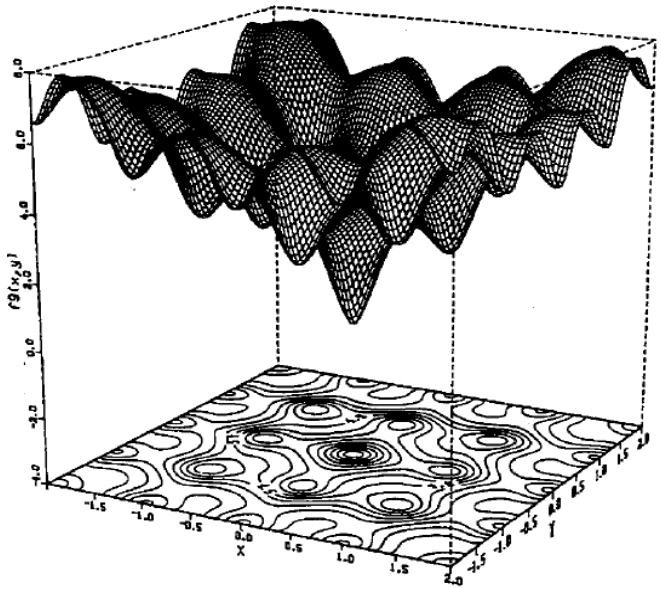
Real valued problems

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

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

$$-exp\left(\frac{1}{n} \cdot \sum_{i=1}^{n} cos(c_3 \cdot x_i)\right) + c_1 + 1$$

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



Using a GA for Neural Network training in order to increase prediction accuracy (PA)

- S-a stabilit o populaţie de cromozomi suficient de largă (50) pentru a asigura variabilitatea populaţiei
- Fiecare cromozom conţine un număr de gene, acestea reprezentând de fapt o pondere din cadrul reţelei neurale. Genele aparţinând populaţiei iniţiale de cromozomi sunt iniţializate cu valori aleatoare în intervalul [-1,1].
- Algoritmul genetic se desfăşoară după scenariul descris în partea teoretică a lucrării:
  - pentru un număr de generaţii stabilit se repetă paşii următori (aprox. 40)
  - sunt evaluaţi toţi cromozomii (fitness = PA)
  - este aplicat operatorul de încrucişare (Crossover)
  - este aplicat operatorul de mutaţie (Mutation)
  - cel mai performant cromozom din generaţia finală va conţine prin genele sale ponderile reţelei antrenate static

GA for Neural Network training – software implementation

Clasa ***CGeneticLearn*** - conţine întregul mecanism pentru antrenarea unei reţele neurale folosind un *algoritm genetic*. Este nevoie de o populaţie de 50 de cromozomi, fiecare cromozom conţinând un număr de gene. Aceste informaţii sunt stocate în vectorul bidimensional *gene*. De asemenea, au fost implementate funcţii pentru fiecare operaţie specifică algoritmilor genetici:

- pentru evaluarea cromozomilor funcţia *calcEvals()*. În cadrul acestei funcţii **genele corespunzătoare fiecărui cromozom sunt introduse ca ponderi într-o reţea neurală** în vedere predicţionării salturilor dintr-un trace. Rezultatul, *acurateţea predicţiei*, va constitui evaluarea cromozomului respectiv. Valorile obţinute în urma evaluărilor vor fi stocate în vectorul *Eval (pentru supravietuirea elitista)*.
- pentru **încrucişarea a doi cromozomi** funcţia ***Reproduce***(), pentru încrucişarea tuturor cromozomilor funcţia ***Cross()*** şi ***Mutate()*** pentru efectuarea de mutaţii.
- În final este implementată şi funcţia ***Run()*** în care pentru un număr de generaţii, stabilit de utilizator, sunt calculate evaluările cromozomilor (*calcEvals()*), cromozomii sunt încrucişaţi (*Cross()*), şi apoi li se aplică operatorul genetic de mutaţie (*Mutate()*). Înainte de a se încheia funcţia *Run()* este **selectat cromozomul cel mai bun şi genele sale aplicate în locul ponderilor reţelei neurale** (*m_nntw→SetPonds(gene[index])*).

GA for Neural Network training – software implementation (class)

```
#define NR_CRO 50
class CGeneticLearn : public CLearningAlg {
private:
...
        long unsigned HRG;
        long unsigned HRLTable[1024];
        int nrno_l1;            int nrno_l2; // number of neurons on Level 1 and Level 2
        int nrp_l1;             int nrp_l2;  // number of wights on Level 1 and Level 2
        double gene[NR_CRO][500];
        int nrGen;
        void Simulate(CString file);
        void Reproduce(int i1,int i2,int j1, int j2);
        void Cross();
        void Mutate();
        void calcEvals();
        void Init();
public:
        CString str;
        CGeneticLearn(CNetw* m_nntw,CProjView* view,CString file,bool* done);
        virtual ~CGeneticLearn();
        void Run();
};
```

Mapping real values on bit strings

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

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

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

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

Floating point mutations 1

General scheme of floating point mutations

$$\bar{x} = \langle x_1, \ ..., x_l \rangle \rightarrow \bar{x}' = \langle x_1', ..., x_l' \rangle$$

$$x_i, x_i' \in [LB_i, UB_i]$$

- **Uniform mutation:**

  x_i' drawn randomly (uniform) from $[LB_i, UB_i]$

- A mutation operator that replaces the value of the chosen gene with a uniform random value selected between the user-specified upper and lower bounds for that gene. This mutation operator **can only be used for integer and float genes**.
- Analogous to bit-flipping (binary) or random resetting (integers)

Floating point mutations 2

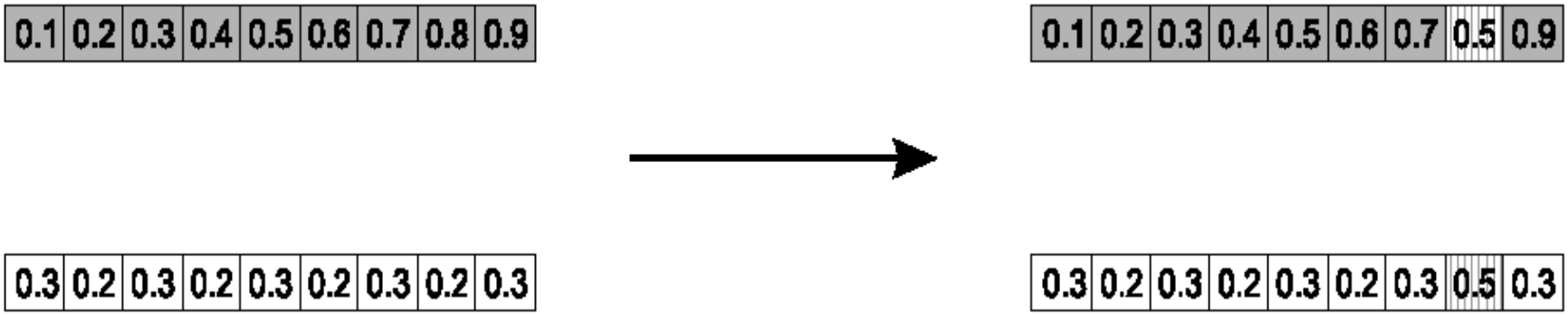
- **Non-uniform mutations**:
  - Many methods proposed,such as time-varying range of change etc.
  - Most schemes are probabilistic but usually only make a small change to value
  - Most common method is to add random deviate to each variable separately, taken from N(0, σ) Gaussian distribution and then curtail to range
  - Standard deviation σ controls amount of change (2/3 of deviations will lie in range (- σ to + σ)

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 (or *convex*)
  - exploits idea of creating children "between" parents (hence a.k.a. *arithmetic* recombination)
  - $z_i = \alpha x_i + (1 - \alpha) y_i$ where α : $0 \leq \alpha \leq 1$.
  - 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 $\alpha = 0.5$

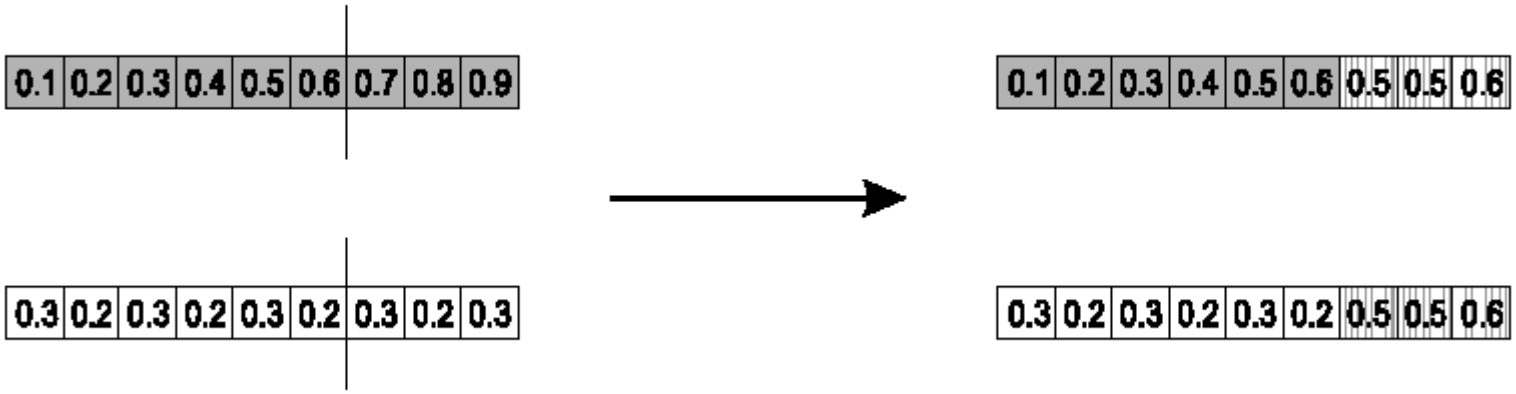


Simple arithmetic crossover

- 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 $\alpha = 0.5$

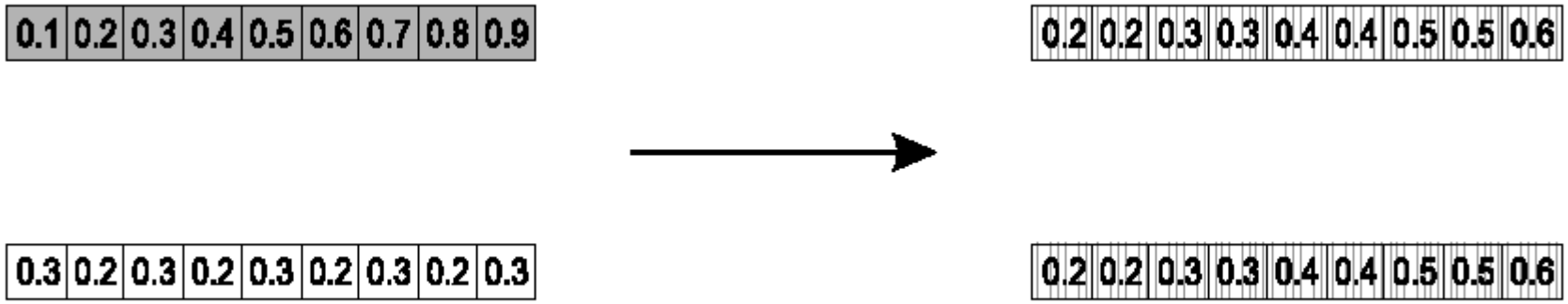


Whole arithmetic crossover

- 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 a = 0.5



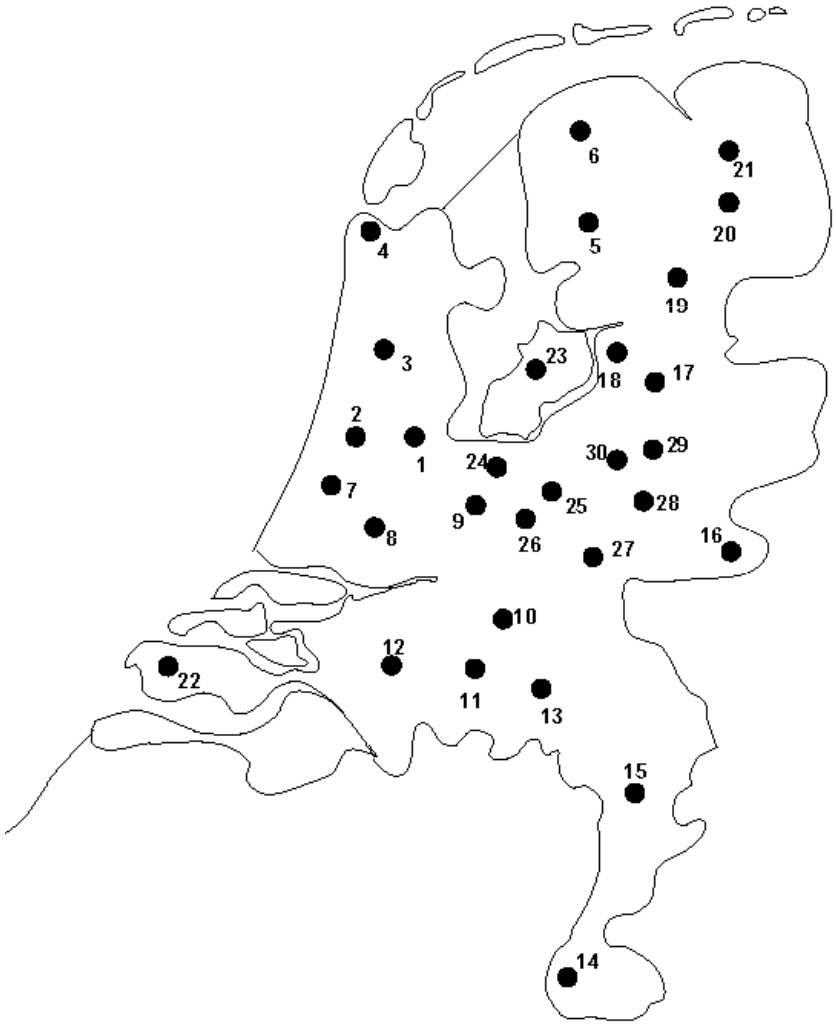
Permutation Representations

- 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

Permutation representation: TSP example

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

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



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 adjacency information

Swap mutation for permutations

- Pick two alleles at random and swap their positions
- Preserves most of adjacency 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 adjacency information (only breaks two links) but disruptive of order information

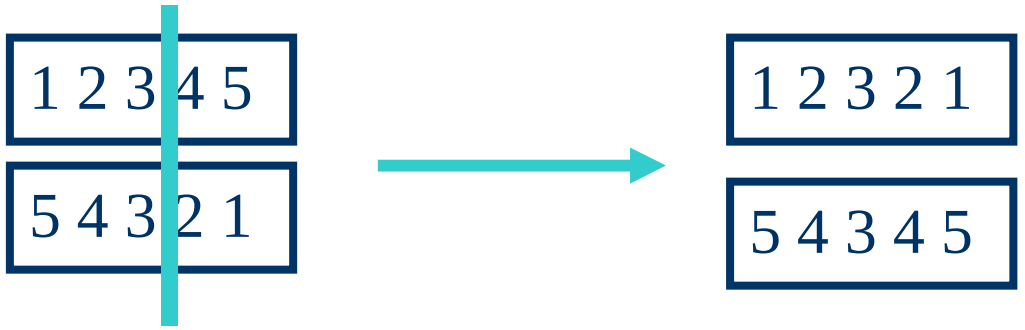
Scramble mutation for permutations

- 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



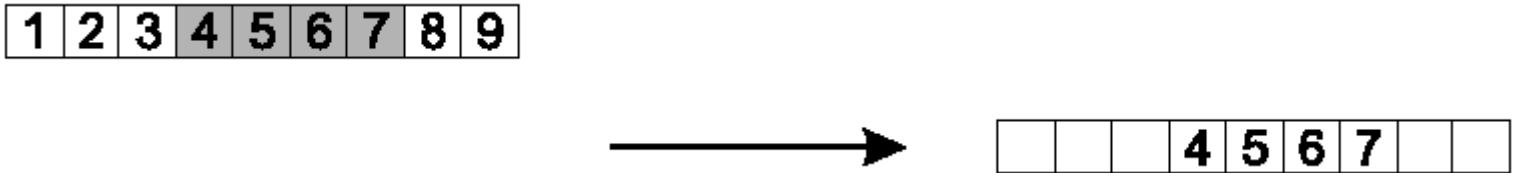
- Many specialised operators have been devised which focus on combining order or adjacency information from the two parents

Order 1 crossover

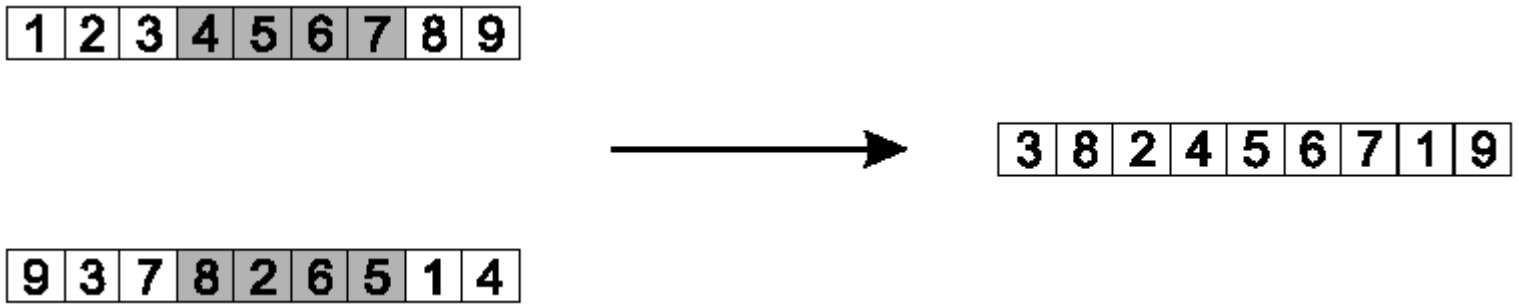
- Idea is to preserve relative order that elements occur
- Informal procedure:
  1. Choose an arbitrary part from the first parent
  2. Copy this part to the first child
  3. Copy the numbers that are not in the first part, to the first child:
     - starting right from cut point of the copied part,
     - 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



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



PMX (Partially matched crossover)

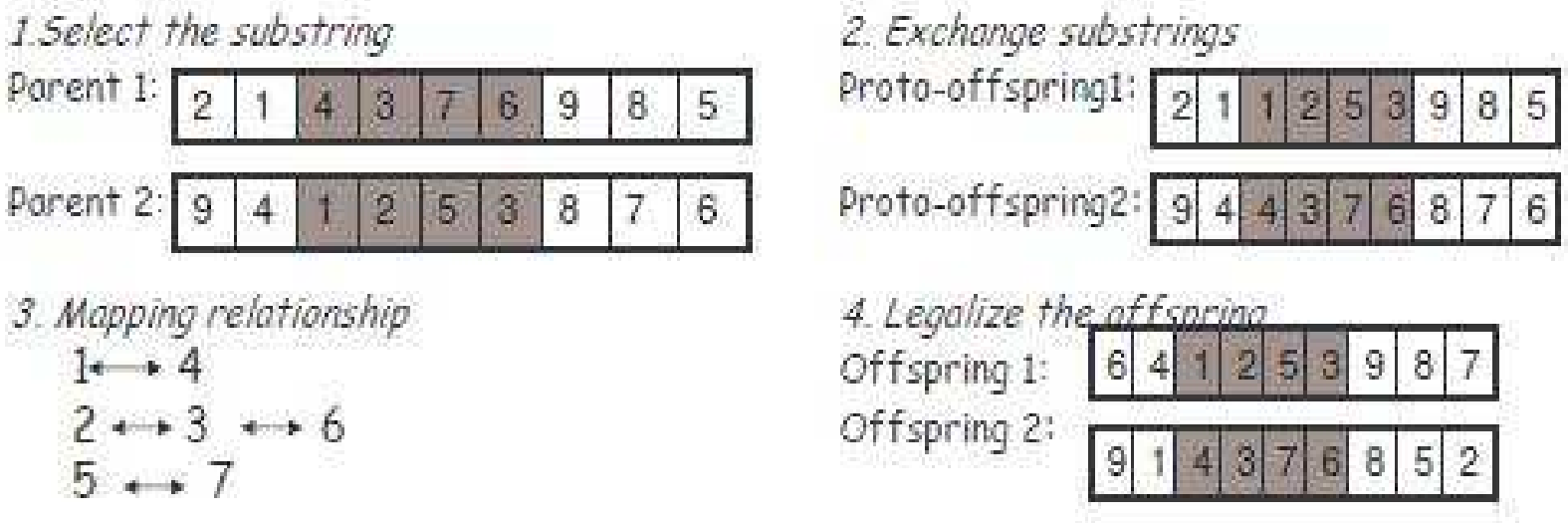
PMX may be viewed as a crossover of permutations, that quarantees that all positions are found exactly once in each offspring.

PMX proceeds as follows:

1. **The two chromosomes are aligned.**
2. **Two crossing sites are selected uniformly at random along the strings, defining a matching section.**
3. **The matching section is used to effect a cross through position-by-position exchange operation.**
4. **Alleles are moved to their new positions in the offspring.**

PMX Algorithm

- **Select a substring uniformly in two parents at random.**
- **Exchange these two substrings to produce proto-offspring.**
- **Determine the mapping relationship according to these two substrings.**
- **Legalize proto-offspring with the mapping relationship.**



Partially Mapped Crossover (PMX)

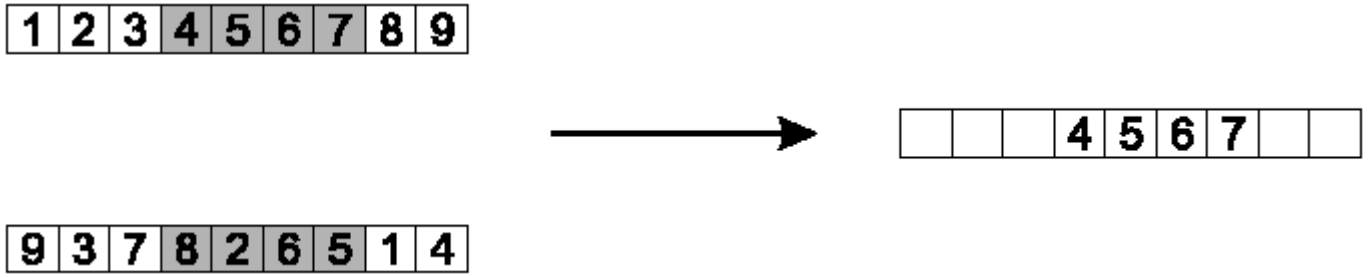
Informal procedure for parents P1 and P2:

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.

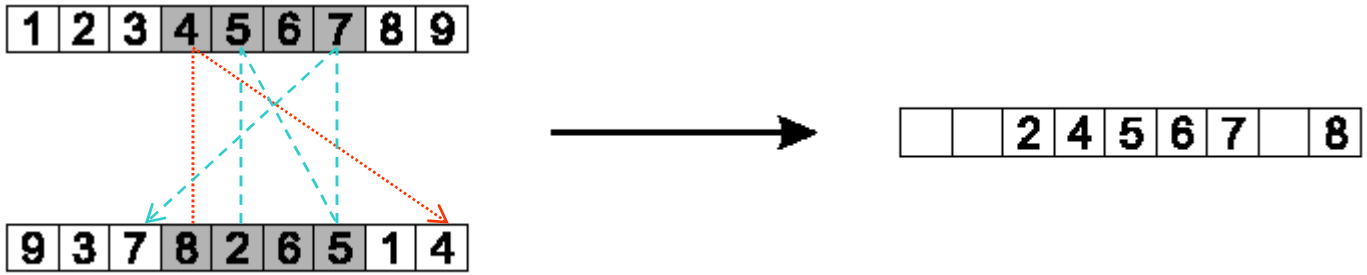
Second child is created analogously

PMX example

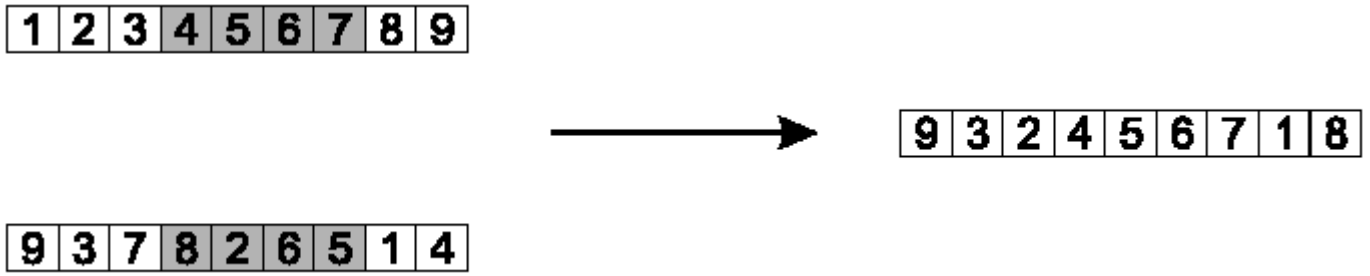
- Step 1



- Step 2



- Step 3



Cycle crossover

Basic idea:

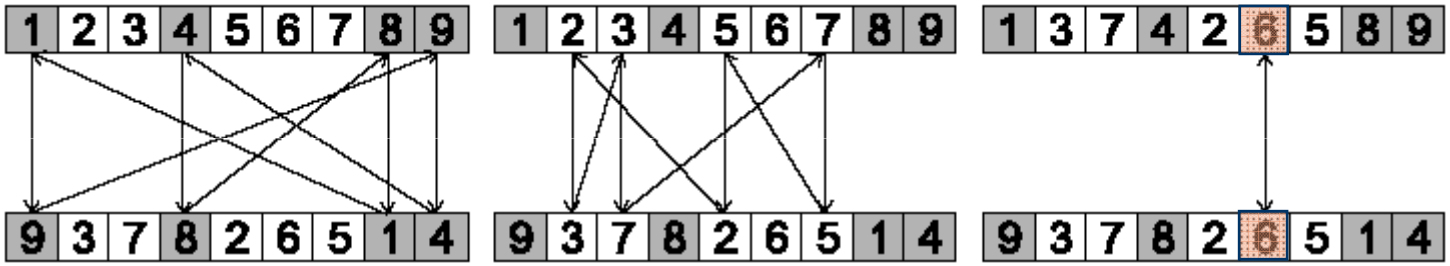
Each allele comes from one parent *together with its position*.

Informal procedure:

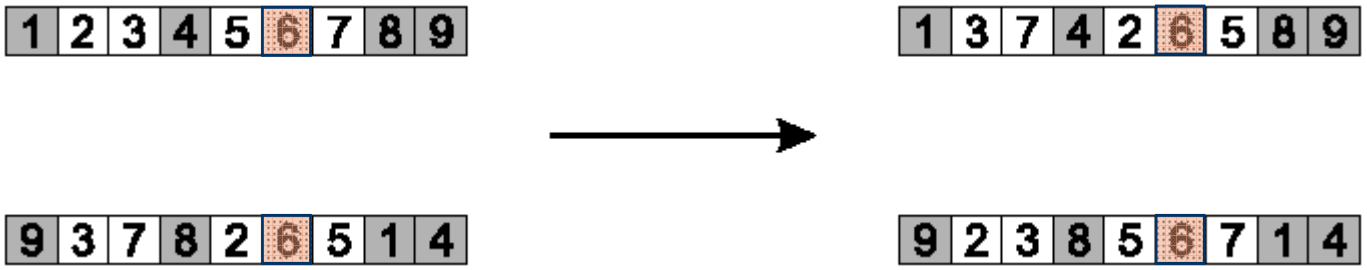
1. Make a cycle of alleles from P1 in the following way.
   (a) Start with the first allele of P1.
   (b) Look at the allele at the *same position* in P2.
   (c) Go to the position with the *same allele* in P1.
   (d) Add this allele to the cycle.
   (e) Repeat step b through d until you arrive at the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



Edge Recombination

- 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 Recombination 2

Informal procedure 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*]

Multiparent recombination

- Recall that we are not constricted 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
- Three main types:
  - Based on allele frequencies, e.g., p-sexual voting generalising uniform crossover
  - Based on segmentation and recombination of the parents, e.g., diagonal crossover generalising n-point crossover
  - Based on numerical operations on real-valued alleles, e.g., center of mass crossover, generalising arithmetic recombination operators

Population Models

- SGA uses a Generational model:
  - each individual survives for exactly one generation
  - the entire set of parents is replaced by the offspring
- At the other end of the scale are Steady-State models:
  - one offspring is generated per generation,
  - one member of population replaced,
- Generation Gap
  - the proportion of the population replaced
  - 1.0 for GGA, 1/pop_size for SSGA

Fitness Based Competition

- 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)
- Selection operators work on whole individual
  - i.e. they are representation-independent
- Distinction between selection
  - operators: define selection probabilities
  - algorithms: define how probabilities are implemented

Implementation example: SGA

- Expected number of copies of an individual *i*

$$E(n_i) = f(i) / \langle f \rangle$$

(μ = pop.size, f(i) = fitness of *i*, ⟨f⟩ avg. fitness in pop.)

$\langle f \rangle = \Sigma f(i) / \mu$

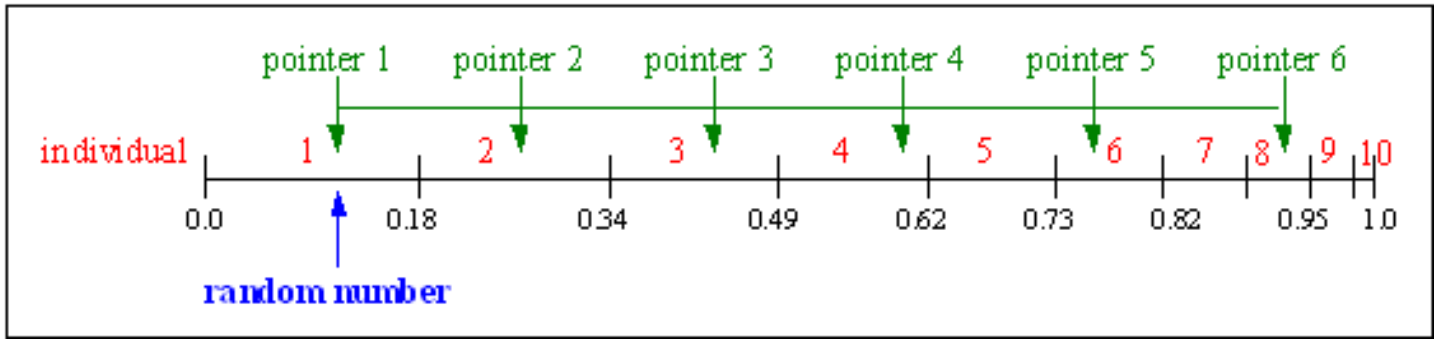
- Roulette wheel algorithm:
  - Given a probability distribution, spin a 1-armed wheel *n* times to make *n* selections
  - No guarantees on actual value of n_i
- Baker's SUS algorithm (**Stochastic universal sampling**):
  - *n* evenly spaced arms on wheel and spin once
  - Guarantees $floor(E(n_i)) \leq n_i \leq ceil(E(n_i))$

SUS algorithm (stochastic universal sampling)

- Stochastic universal sampling provides **zero bias** and **minimum spread**.
- The individuals are mapped to contiguous segments of a line, such that each individual's segment is equal in size to its fitness exactly as in roulette-wheel selection.
- Here equally spaced pointers are placed over the line as many as there are individuals to be selected.
- Consider *NPointer* the number of individuals to be selected, then **the distance between the pointers are 1/*NPointer*** and **the position of the first pointer is given by a randomly generated number in the range [0, 1/*NPointer*].**

SUS algorithm (continued)

- For 6 individuals to be selected, the distance between the pointers is 1/6=0.167.
- Sample of 1 random number in the range [0, 0.167]: (e.g. *0.1)*

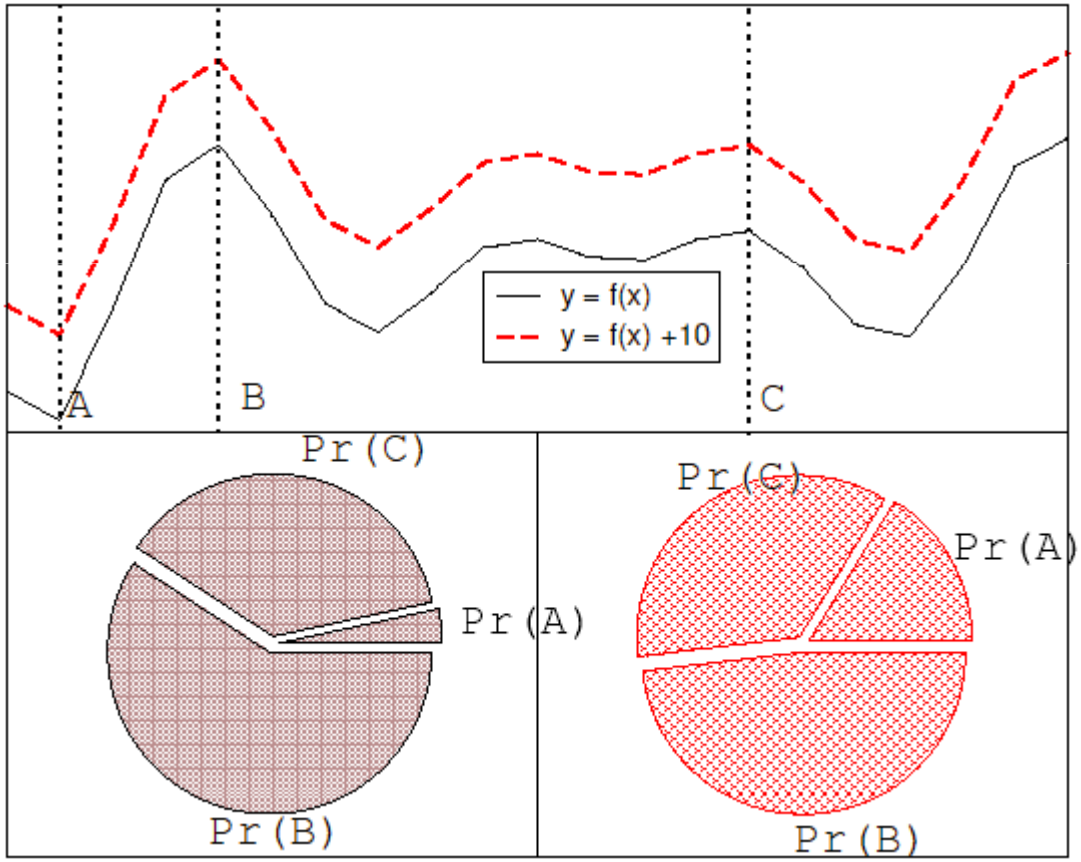


- After selection the mating population consists of the individuals: 1, 2, 3, 4, 6, 8.
- SUS ensures a selection of offspring which is closer to what is deserved then roulette wheel selection.

Fitness-Proportionate Selection

- Problems include
  - One highly fit member can rapidly take over if rest of population is much less fit: Premature Convergence
  - At end of runs when fitnesses are similar, lose selection pressure
  - Highly susceptible to function transposition
- Scaling can fix last two problems
  - Windowing: $f'(i) = f(i) - \beta^t$
    - where β is worst fitness in this (last n) generations
  - Sigma Scaling: $f'(i) = max(f(i) - (\langle f \rangle - c \cdot \sigma_f), 0.0)$
    - where c is a constant, usually 2.0

Function transposition for FPS



Rank – Based Selection

- Attempt to remove problems of FPS by basing selection probabilities on *relative* rather than *absolute* fitness
- Rank population according to fitness and then base selection probabilities on rank where fittest has rank μ *(or n – number of individuals)* and worst rank 1
- This imposes a sorting overhead on the algorithm, but this is usually negligible compared to the fitness evaluation time

Linear Ranking

$$p_i = \frac{1}{n}\left[s - \frac{2x(r_i - 1)x(s-1)}{n-1}\right]$$

- Parameterised by factor *s:* $1.0 < s \leq 2.0$
  - measures advantage of best individual
  - r_i the rank of individual *i (descending sorting).*
  - in GGA this is the number of children allotted to it
- 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	2	0.5	0.67	0.5
C	4	2	0.4	0.33	0.33
Sum	10		1.0	1.0	1.0

Exponential Ranking

$$P_{exp-rank}(i) = \frac{1 - e^{-i}}{c}.$$

- Linear Ranking is limited to selection pressure
- Exponential Ranking can allocate more than 2 copies to fittest individual
- Normalise constant factor *c* according to population size

Binary tournament

Binary tournament is run to determine a relative fitness ranking. Initially the entire population is in the tournament.

Two members are selected at random to compete against each other with only the winner of the competition progressing to the next level of the tournament .

When the tournament is over, the relative fitness of each member of the population is awarded according to the level of the tournament it has reached.

Tournament Selection

- All methods above rely on global population statistics
  - Could be a bottleneck esp. on parallel machines
  - Relies on presence of external fitness function which might not exist: e.g. evolving game players
- Informal Procedure:
  - Pick *k* members at random then select the best of these
  - Repeat to select more individuals

Tournament Selection 2

- Probability of selecting *i* will depend on:
  - Rank of *i*
  - Size of sample *k*
    - higher *k* increases selection pressure
  - Whether contestants are picked with replacement
    - Picking without replacement increases selection pressure
  - Whether fittest contestant always wins (deterministic) or this happens with probability *p*
- For $k = 2$, time for fittest individual to take over population is the same as linear ranking with $s = 2 \bullet p$

Survivor Selection

- Most of methods above used for parent selection
- Survivor selection can be divided into two approaches:
  - Age-Based Selection
    - e.g. SGA
    - In SSGA can implement as “delete-random” (not recommended) or as first-in-first-out (a.k.a. delete-oldest)
  - Fitness-Based Selection
    - Using one of the methods above or

Two Special Cases

- Elitism
  - Widely used in both population models (GGA, SSGA)
  - Always keep at least one copy of the fittest solution so far
- GENITOR: a.k.a. “delete-worst”
  - From Whitley’s original Steady-State algorithm (he also used linear ranking for parent selection)
  - Rapid takeover : use with large populations or “no duplicates” policy

Example application of order based GAs: JSSP

Precedence constrained **job shop scheduling problem**

- J is a set of jobs.
- O is a set of operations
- M is a set of machines
- *Able* $\subseteq$ O $\times$ M defines which machines can perform which operations
- *Pre* $\subseteq$ O $\times$ O defines which operation should precede which
- *Dur* : $\subseteq$ O $\times$ M $\rightarrow$ IR defines the duration of o $\in$ O on m $\in$ M

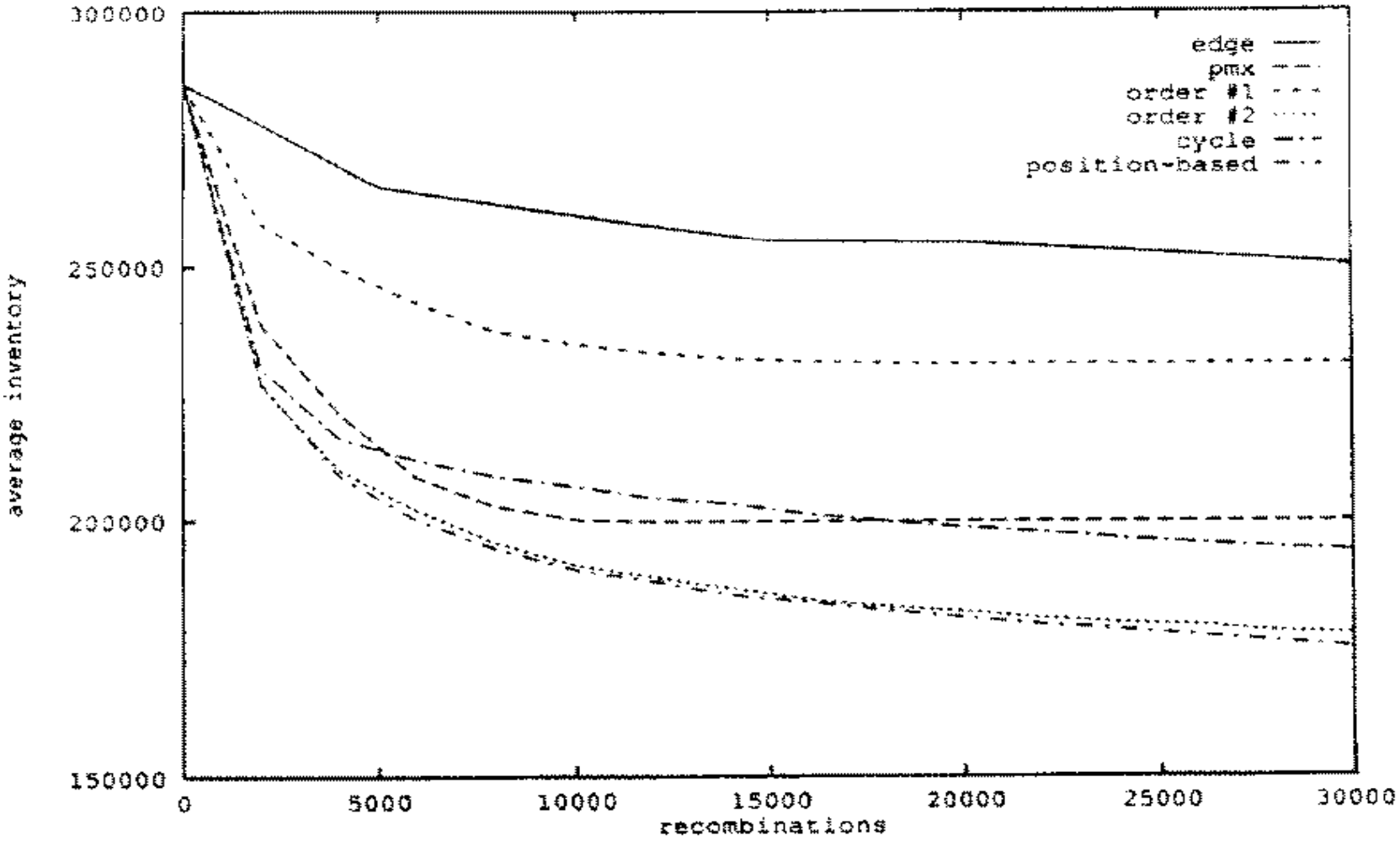
The goal is now to find a schedule that is:

- Complete: all jobs are scheduled
- Correct: all conditions defined by *Able* and *Pre* are satisfied
- Optimal: the total duration of the schedule is minimal

Precedence constrained job shop scheduling GA

- Representation: individuals are permutations of operations
- Permutations are decoded to schedules by a decoding procedure
  - take the first (next) operation from the individual
  - look up its machine (here we assume there is only one)
  - assign the earliest possible starting time on this machine, subject to
    - machine occupation
    - precedence relations holding for this operation in the schedule created so far
- fitness of a permutation is the duration of the corresponding schedule (to be minimized)
- use any suitable mutation and crossover
- use roulette wheel parent selection on inverse fitness
- Generational GA model for survivor selection
- use random initialisation

JSSP example: operator comparison



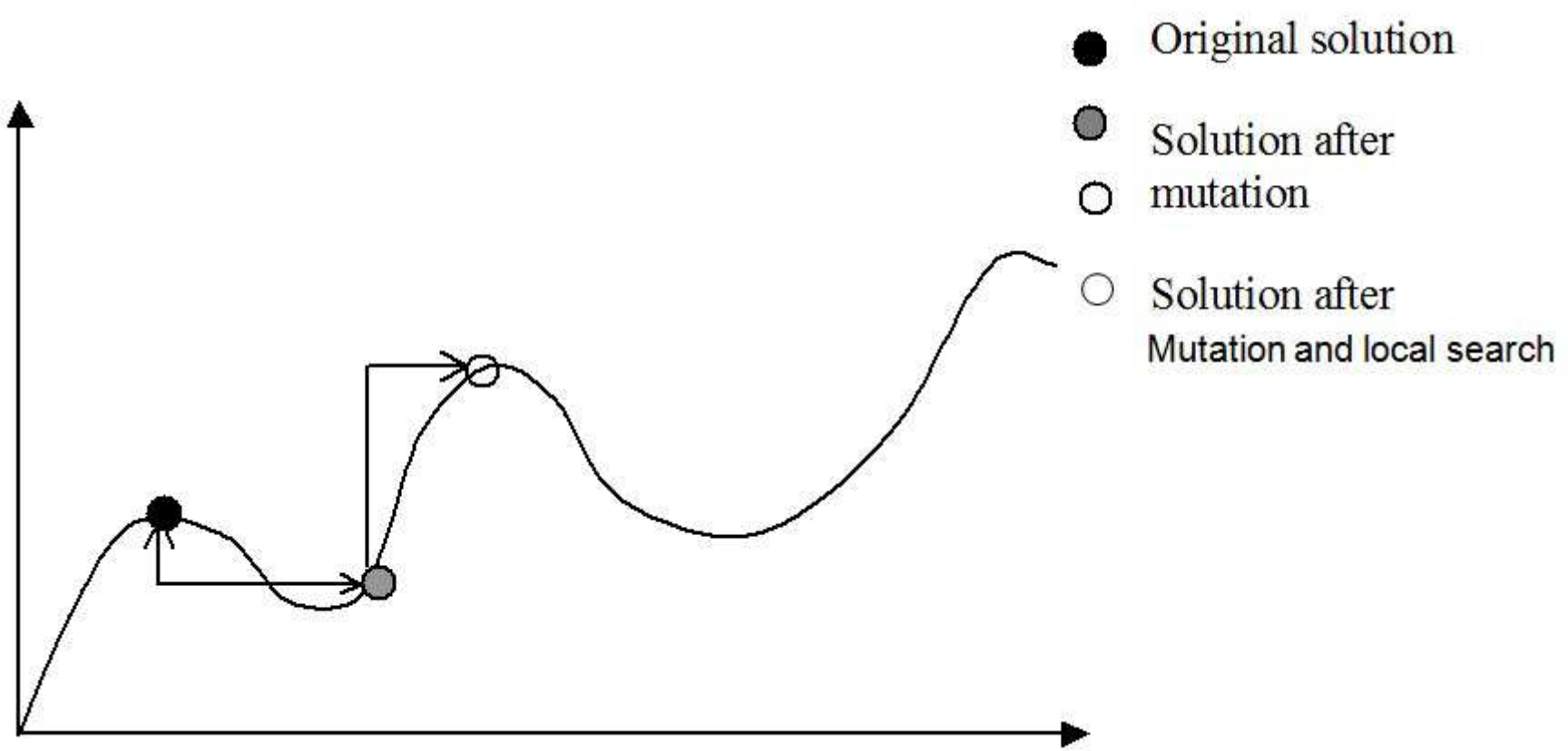
MEMETIC ALGORITHMS

- Memetic algorithm introduced by Moscato and Norman in 1992 is an improved variant of genetic algorithm.
- It takes the concept of evolution as in genetic algorithm.
- However, while genetic algorithm is based on biological evolution, memetic algorithm is based on *cultural* evolution or *idea* evolution. In the evolution of ideas, an idea may be improved not only by recombination from others, but also by adaptation from itself.

A *meme*

- A meme, a unit of information in memetic algorithm can be improved by the individual holding it before it is passed on.
- A meme differs from a gene in that as it is passed between individuals, each individual adapts the meme as it see best whereas genes are passed unchanged.
- A basic memetic algorithm, is then, an *evolution algorithm* incorporated with some *local search* technique.

An Example Memetic Operation



Memetic Algorithm

- Normally, the local search technique is *hill-climbing* and the evolutionary operators are only mutation operators.
- In genetic algorithm, while the mutation creates new genes for the population, the crossover operator orients seeking the best solution from the genes in the population.
- In memetic algorithm, this orientation is achieved by local search. Local search reduces the search space and reaches to high quality solution faster.

Optimized Simulated Annealing for Network-on-Chip Application Mapping

- Simulating Annealing (SA) is a tree search technique that combines hill climbing with a random walk in order to yield efficiency and also completeness. The hill climbing algorithm never makes downhill moves so it is guaranteed to be incomplete (**because it can get stuck in a local maximum**). On the other hand, moving to a successor tree node, chosen randomly from all the possible successors, is very inefficient but more complete. **Simulated Annealing reduces to Hill Climbing**.
- The idea behind this algorithm comes from metallurgy, where annealing is the process used to temper or harden metals and glass by heating them to a high temperature and then gradually cooling them, thus allowing the material to coalesce into a low-energy crystalline state.
- Using **Optimized Simulated Annealing algorithm as a mutation operator** it obtain a hybrid algorithm: an Evolutionary Algorithm which incorporates a Simulated Annealing technique [Ciprian Radu].
- OSA performs a context-aware mapping and outputs two cores which must be swapped.
- Using domain-knowledge, the Optimized Simulated Annealing (OSA) algorithm performs a dynamic and implicit core clustering and limits the number of iterations per annealing temperature based on the given application and network.

Outline of a Memetic Algorithm

- create initial population
- repeat
  1. take each individual in turn:
     Choose a mutation method
     Apply mutation operator to chosen individuals
     Apply hill-climbing to individual just created.
     Insert it into the population.
  2. select a half of them to reduce the population to its original size.

  until termination condition is true

Memetic Algorithms have been shown **to be orders of magnitude faster and more accurate than EAs on some problems,** and are **the "state of the art" on many problems**