Biologically Inspired Computing:

Operators for Evolutionary Algorithms

Some basic operators

How to think of genetic operators

Operators **provide our means of generating new candidate solutions.**

- *We want operators to have a fair chance of yielding good new solutions (small change, and/or combine bits from solutions we already know are good)*
- *We also (obviously) want to be able to potentially explore the whole space.*

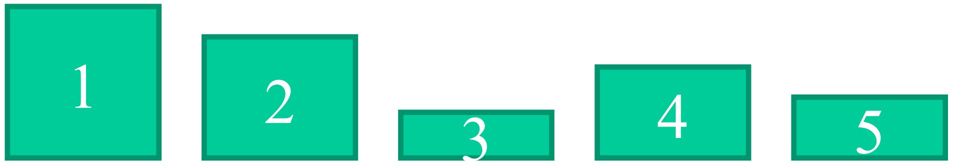
Example of an encoding (and associated operators)

Bin-Packing

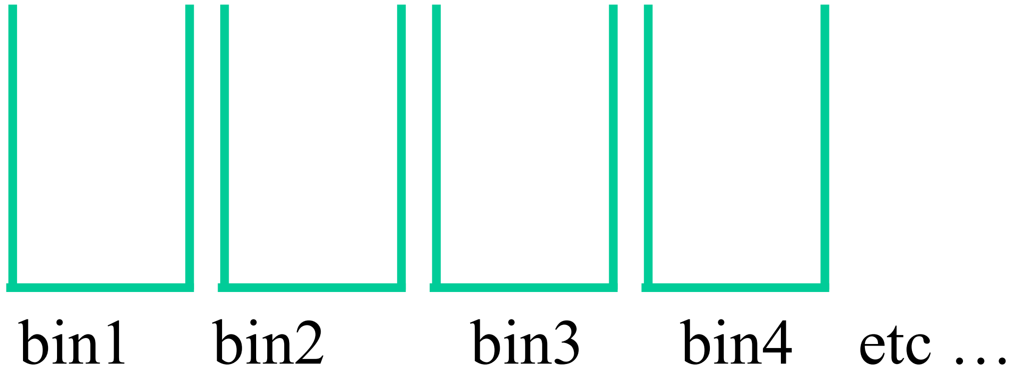
The Bin-Packing Problem:

You have items of different sizes or weights:

e.g: 1 (30kg) 2 (25kg) 3 (10kg) 4 (20kg) 5 (15kg)



And you have several `bins' with a max capacity, say 50kg.



Find a way to pack them into the smallest number of bins. This is, *in general*, a **hard** problem.

A simple encoding for bin-packing

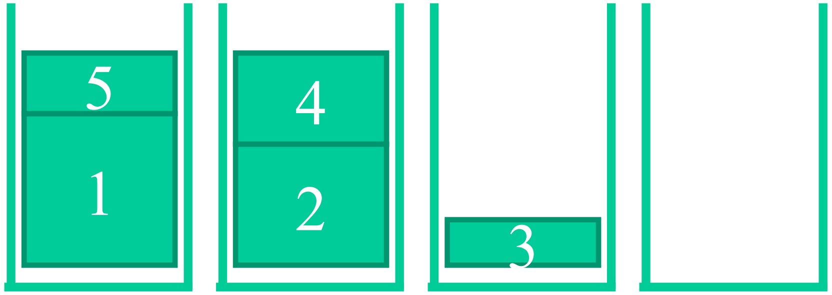
Suppose we have *L* items, and we have a maximum of k bins available,

A simple encoding is as follows:

- A solution is represented by a list of *L* integers, each of them being any number from 1 to k inclusive.
- the meaning of such a solution, like for example '**3**, **1**, 2, **1**, **2** ….'
  is 'the 1st item is in bin 3, the 2nd item is in bin 1, **the 3rd item is in bin 2**, the **4th item is in bin 1**, **the 5th item is in bin 2**, [and so on …]'

So, in our simple example,

1,2,3,2,1 encodes this solution



Genetic Operators / Variation Methods

Often we have used a ***k*-ary encoding**, in which a candidate solution is just a list of ***L*** numbers, each of which can be anything from 0 to k-1 (or 1 to k).

E.g. a simple bin-packing representation that uses a k-ary encoding. These might be candidate solutions from a k-ary encoding with $L = 10$ and $k = 20$:

[17, 2, 19, 1, 1, 1, 5, 11, 12, 2]

[16, 19, 2, 19, 2, 3, 4, 7, 5, 2]

Mutation in *k*-ary encodings

Single-gene new-allele mutation:

Choose a gene at random, and change it to a random **new** value. E.g. 352872 → 312872

***M*-random-gene new-allele mutation:**

Repeat the above *M* times. For example, in a 1000-gene problem, we might use 5-gene mutation. This will result in anything from 1 to 5 new gene values.

Mutation in *k*-ary encodings

Single-gene random-allele mutation:

Choose a gene at random, and change it to a random value. This is the same as **single-gene new-allele** mutation, except that it doesn't take care to make sure we have a new value for the gene. So, often (especially if *k* is small) it will lead to no change at all. But that's not a problem – in the EA context, it means that the next generation contains **an extra copy** of an individual that survived selection (so is probably quite good), and in fact it might not be in the new population otherwise.

***M*-distinct-gene new-allele mutation:**

Same as **M-random-gene**, but making sure that *M* *different* genes are changed.

Genewise mutation with strength *m;*

Go through the solution gene by gene, and mutate every one of them with probability *m*. E.g. if *L* = 100 we might use *m* = 0.01 – usually, just one gene will get changed, but possibly none, and possibly 2 or more. There is even a tiny chance of all of them being changed.

Single-gene mutation:

Choose a gene at random, and add a small random deviation to it. Often chosen from a Gaussian distribution.

Vector mutation:

Generate a small random vector of length *L*, and add the deviation to these positions in the chromosome.

Crossover in *k*-ary encodings

One-Point Crossover.

For encodings of length L, one point crossover works as follows:

1. Choose a crossover point k randomly (a number from 1 to L-1)
2. The first k genes of the child will be the same as the first k genes of parent 1
3. The genes of the child from gene k+1 onwards will be the same as those of parent 2.

Crossover in *k*-ary encodings

Two-Point Crossover.

For encodings of length L, two point crossover works as follows:

1. Choose two crossover points j and k randomly, making sure that $1 <= j < k < L$
2. Genes 1 to j of the child will be the same as genes 1 to j of parent 1;
3. genes j+1 to k of the child will be the same as genes j to k of parent 2.
4. the remainder of the child will be the same as parent 1.

1-point example:

Parent1: **1, 3, 4, 3, 6, 1, 3, 6, 7, 3, 1, 4**

Parent2: **3, 5, 2, 6, 7, 1, 2, 5, 4, 2, 2, 8**

Random choice of $k = 6$

Child: **1, 3, 4, 3, 6, 1, 2, 5, 4, 2, 2, 8**

2-point example:

Parent1: **1, 3, 4, 3, 6, 1, 3, 6, 7, 3, 1, 4**

Parent2: **3, 5, 2, 6, 7, 1, 2, 5, 4, 2, 2, 8**

Random choices: $j = 3$, $k = 10$

Child: **1, 3, 4, 6, 7, 1, 2, 5, 4, 2, 1, 4**

Uniform Crossover

For encodings of length *L*, Uniform crossover works as follows:

```
For each gene $i$ from 1 to $L$
        Let $c$ = 1 or 2, with equal probability.
        Make Gene $i$ of the child the same as gene $i$ of Parent $c$
```

In other words, we simply create the child one gene at a time, flipping a coin each time to decide which parent to take the gene from.

Uniform crossover example:

Parent1: **1, 3, 4, 3, 6, 1, 3, 6, 7, 3, 1, 4**

Parent2: **3, 5, 2, 6, 7, 1, 2, 5, 4, 2, 2, 8**

If our random choices in order were:

121221121221

the child would be:

Child: **1, 5, 4, 6, 7, 1, 3, 5, 7, 2, 2, 4**

Operators in order-based encodings

Often, our chosen encoding for a problem will be a ***permutation***. This is the usual encoding for problems such as the travelling salesperson problem, and many others.

Clearly, *k*-ary encoding operators are invalid in this case. The following slides show common operators that can be used.

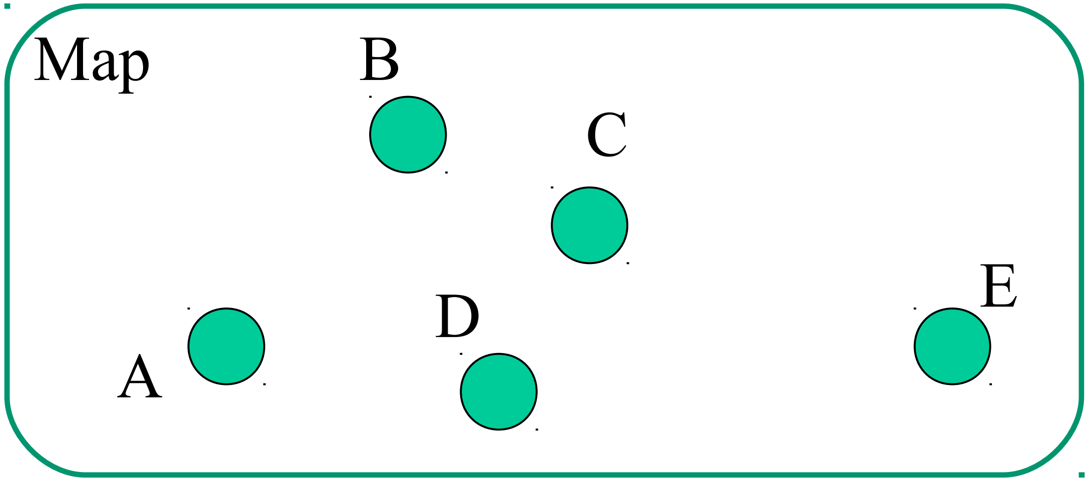
Example of an encoding (and associated operators)

The TSP

The Travelling Salesperson Problem

You have *N* 'cities', and an associated distance matrix. Find the shortest **tour** through the cities. I.e., starting from one of the cities ('A', for example), what is the quickest way to visit all of the other cities, and end up back at 'A' ?

Many many real problems are based on this (vehicle routing problems, bus or train scheduling), or exactly this (drilling thousands of holes in a metal plate, or … an actual Travellling Salesperson problem!).



	A	B	C	D	E
A		5	7	4	15
B	5		3	4	10
C	7	3		2	7
D	4	4	2		9
E	15	10	7	9	

Mutation in order-based encodings

In all examples we will assume our encoding is a permutation of L items, where $L = 10$.

swap mutation is simply this:

Choose randomly two distinct genes, i and j, and then swap the genes at these positions.

Adjacent-swap mutation is simply this:

Choose any gene j randomly, then swap it with gene $j+1$ (modulo L).

Swap mutation example:

Parent: A C J I B D E G H F

Random choices: $i = 3$; $j = 7$

Child: A C E I B D J G H F

Adjacent-swap mutation example

Parent: A C J I B D E G H F

Random choice: $j = 8$

Child: A C J I B D E H G F

k-Inversion-mutation

Choose a random *chunk* of the parent of size k. That is, choose a random gene j from 1 to L, and the chunk is the section from genes j *to* $j+k-1$ (modulo L) inclusive. Reverse the genes in this chunk.

(note: when $k = 2$, what is this identical to?)

Example of *k*-Inversion mutation

Here is 3-inversion mutation applied to:

A B D G F C E

Choose a random chunk of size 3:

A B D G F C E

Reverse the genes in this chunk:

A G D B F C E

Crossover in order-based encodings

There are various sensible ways to produce a valid child that combines aspects of two different permutation parents. What might be best depends very much on the application. We will describe two generic order based operators.

k-gene Order-based Crossover

Works like this:

First, make the child a copy of parent 1.

Next, randomly choose *k* distinct genes of the child.

Next, reorder the **values** of these genes so that they match their order in parent 2.

4-gene order based crossover example.

Parent 1: A C J I B D E G H F

Parent 2:   F E A I H J D B C G

Random choices of 4 distinct genes: 2, 5, 6, 9

Child is initially same as parent 1 – shown here with the random choices highlighted:

Child: A C J I B D E G H F

The order of these four gene values in Parent 2 is: H, D, B, C

So, we impose that ordering on the child, but keeping them in the same positions:

Child: A H J I D B E G C F

k-gene Position-based Crossover

Works like this:

First, make the child a copy of parent 1.

Next, randomly choose *k* distinct **gene positions** of the child. Let *V* be the set of gene values at these positions.

Next, copy the genes of parent 2 that are *not* in *V* into the child, overwriting the child's other genes, in their parent 2 order.

4-gene position based crossover example.

Parent 1: A C J I B D E G H F

Parent 2:   F E A I H J D B C G

Random choices of 4 distinct positions: 2, 5, 6, 9

Child is initially same as parent 1 – shown here with the random positions highlighted:

Child: A C J I B D E G H F

Now let's blank out all of the other genes:

Child: * C * * B D * * H *

We proceed by filling the child up with its missing genes, in the order they appear in Parent 2: This order is: F E A I J G. So the child becomes:

Child: F C E A B D I J H G

Operators for real-valued *k*-ary encodings

Here the chromosome is a string of *k* real numbers, which each may or may not have a fixed range (e.g. between −5 and 5).

e.g. 0.7, 2.8, −1.9, 1.1, 3.4, −4.0, −0.1, −5.0, …

All of the mutation operators for *k*-ary encodings, as previously described in these slides, can be used. But we need to be clear about what it means to randomly change a value. In the previous slides for *k*-ary mutation operators we assumed that a change to a gene mean to produce a random (new) value **anywhere in the range of possible values …**

that's fine … we can do that with a real encoding, but this means we are choosing the new value for a gene uniformly at random.

Mutating a real-valued gene using a uniform distribution

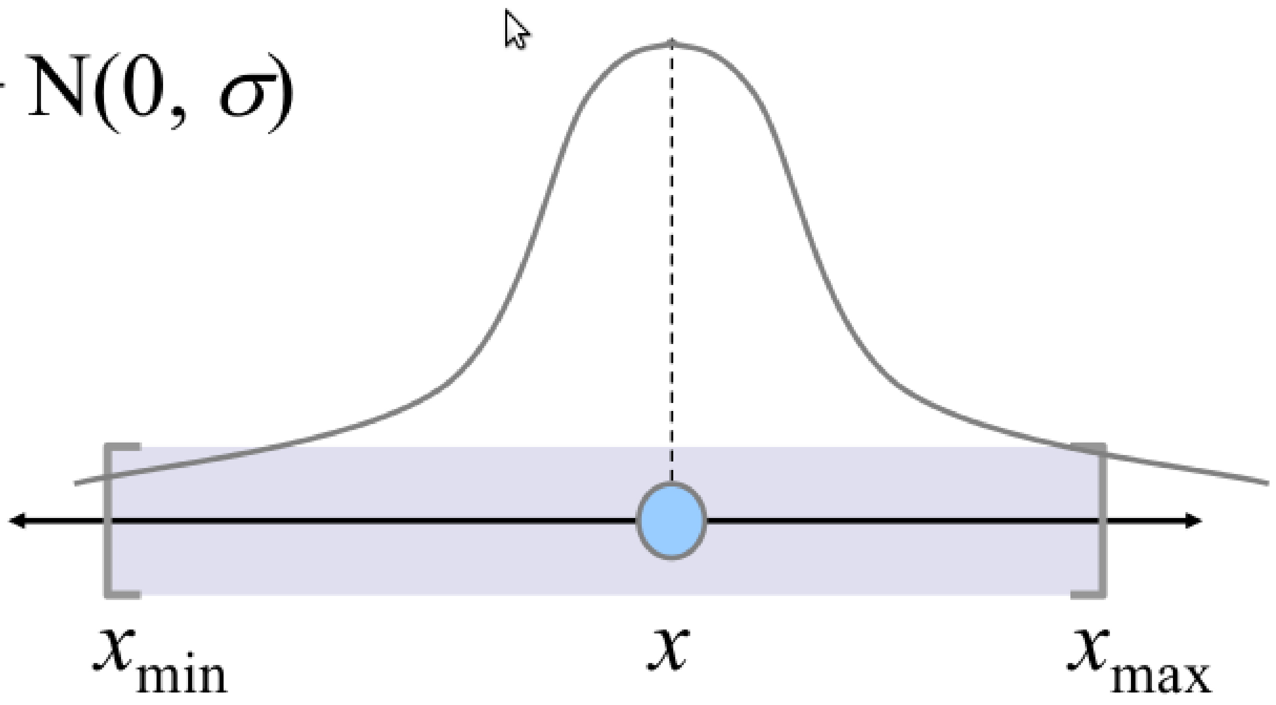
Range of allowed values for gene: 0—10

New value can be anywhere in the range, with any number equally likely.

Normally Distributed Mutation

- Perturb the gene value using a zero-mean Gaussian distribution

$$x_{\text{new}} = x + \text{N}(0, \sigma)$$



Mutation in real-valued encodings

Most common is to use the previously indicated mutation operators (e.g. single-gene, genewise) but with Gaussian perturbations rather than uniformly chosen new values.

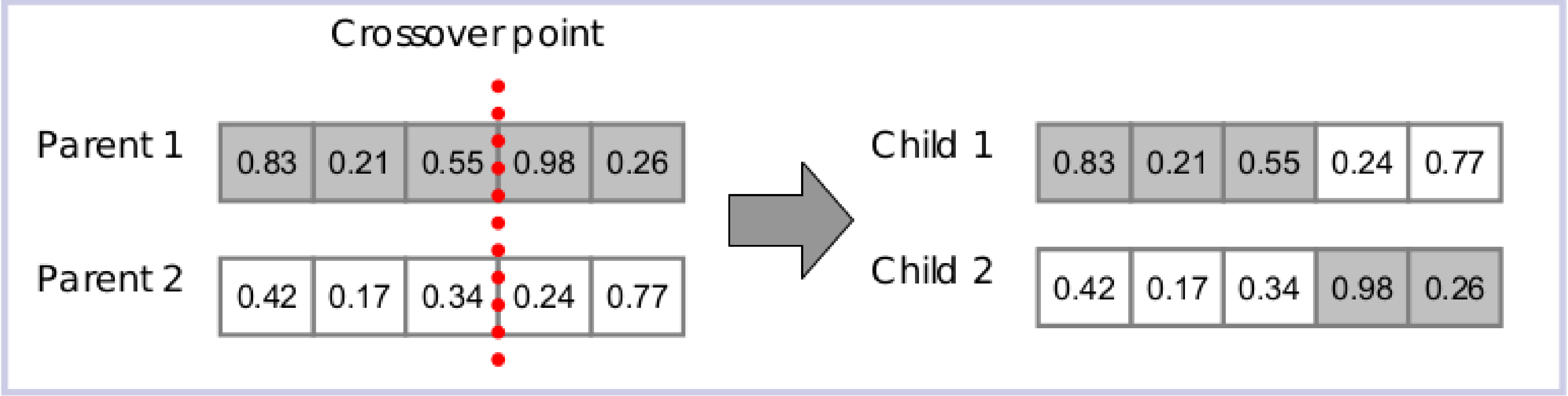
Crossover in real-valued encodings

All of the *k*-ary crossover operators previously described can be used.

But there are other common ones that are only feasible for real-valued encodings:

Single-Point Crossover

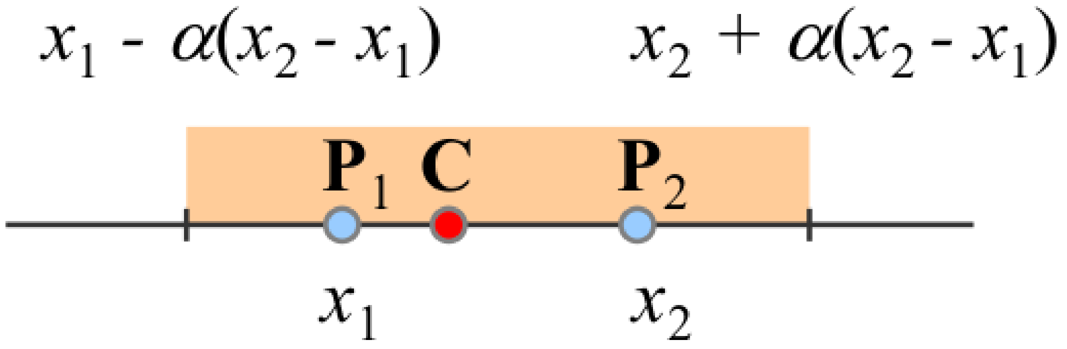
- Similar to the crossover operator used in the binary-coded GAs



- According to the number of crossover points, there are also two-point, three-point and *n*-point crossover

Blend Crossover

- Given the two parents x_1 and x_2 where $x_1 < x_2$, the blend crossover randomly selects a child in the range $[\, x_1-\alpha(x_2-x_1),\ x_2+\alpha(x_2-x_1)]$



- It is often suggested that a good choice of α is 0.5

Blending Crossover

This crossover operator is a kind of linear combination of the two parents that uses the following equations for each gene:

Offspring #1 = parent1 - b * (parent1 - parent2)

Offspring #2 = parent2 + b * (parent1 - parent2)

Were b is a random value between 0 and 1.

Treat the parents like vectors

Types of bend crossover:

- Line crossover
- Box crossover

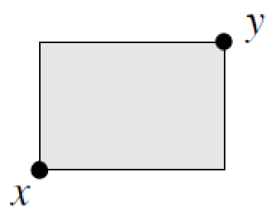
Box and Line crossover: general form

Parent1: x1, x2, x3, …., xN
Parent2: y1, y2, y3, …, yN
given parameter α (typically values are 0, 0.1 or 0.25)
General form:
Child is (($x1 - \alpha$) + **u** (y1 – x1), (($x2 - \alpha$) + **u** (y2 – x2), … etc)
Where: **u** is a uniform random number between 0 and $1+2\alpha$

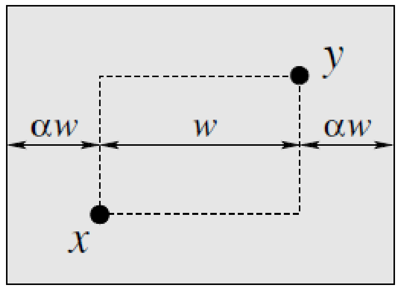
Line crossover: $\alpha = 0$; **u** is generated once for each crossover, and is the same for every gene.
Extended line crossover: $\alpha > 0$, **u** is generated once for each crossover, and is the same for every gene.
Box crossover: $\alpha = 0$; **u** is different for every gene
Extended box crossover: $\alpha > 0$; **u** is different for every gene

Box and Line crossover operators for real-valued chromosomes – figure is from this paper:

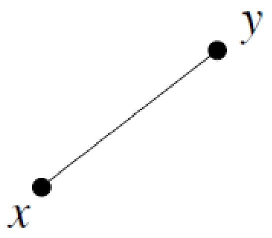
http://cdn.intechopen.com/pdfs/30229/InTech-The_roles_of_crossover_and_mutation_in_real_coded_genetic_algorithms.pdf



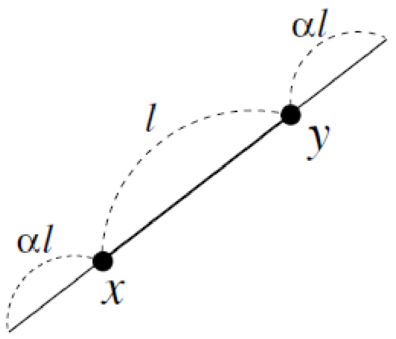
(a) Box crossover



(b) Extended-box crossover



(c) Line crossover



(d) Extended-line crossover

Fig assumes you are crossing over two 2-gene parents, **x** (x1,x2) and **y** (y1,y2)

Box: child is

(x1 + **u1** (y1 – x1), x2 + **u2**(y2 – x2))

Where **u1** and **u2** are uniform random numbers between 0 and 1

Line: child is: **x** + **u** **(y** – **x)**

or (x1+ **u**(y1 – x1) , x2 + **u** (y2 – x2))

Where **u** is a random number between 0 and 1