UiO : Department of Informatics
University of Oslo

Biologically inspired computing - Lecture 3

Representations

(Genetic algorithms & Genetic programming)

This lecture

- Representations
  - Recombination
  - Mutation

Optimization problems

- Continuous optimization
- 0-1 knapsack problem
- Other knapsack problems
- Travelling salesman problem
- Task solving problems

Real-valued representations

- As shown in the previous lecture
  - Represents continuous solution spaces
  - The solution parameters are often accompanied by strategy parameters for adaptive normal distribution-based mutation

0.1	3.3	1.7	3.4	7.2	5.9

Binary representation

- The representation used in the simple genetic algorithm (SGA)
  - Directly inspired by low-level encoding in DNA
  - Uses a binary (0,1) coding instead of the quaternary (G,T,A,C) coding used in nature

0	1	1	0	1	0	0	0

Integer representation

- Each element is directly coded as an integer
  - Usually restricted to some pre-defined ranges

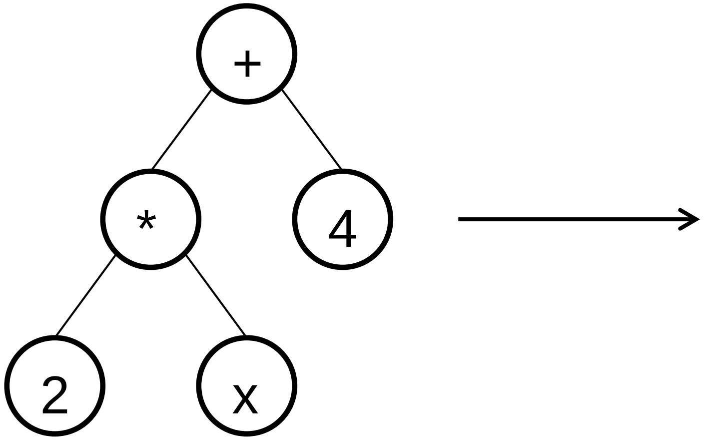
0	5	8	3	1	3	7	5

Permutation representation

- Used to solve problems like the travelling salesman
  - Known set of actions (go to town X)
  - Want to optimize their sequence

Tree representation

- Tree representations of programs or arithmetic expressions
  - Mainly used in genetic programming



$$2x + 4$$

Representations

```
def evolve():
    P.x = initialize_population()
    P.fitness = evaluate(P.x)
    while not_done():
        Q.x = reproduce(P)
        Q.x = mutate(Q.x)
        Q.fitness = evaluate(Q.x)
        P = survival(P,Q)
    return best(P).x
```

- The central concepts in evolutionary algorithms are independent of representation
- Mutation and recombination must be tailored to the representation used

Indirect representations

- Most problems will have a fixed solution representation associated with it

- However, sometimes it is beneficial to evolve solutions using a different representation and then transform them to do the evaluation

Expanding the analogy

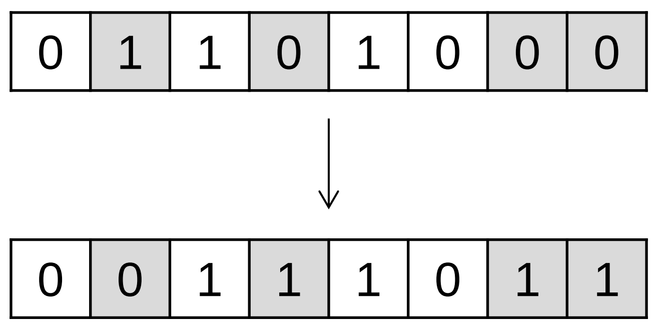
Optimization	Biology
Candidate solution	Individual
Representation used in the EA	Genotype, chromosome
Problem-defined representation	Phenotype
Position/element of the genotype	Locus, gene
Old solution	Parent
New solution	Offspring
Solution quality	Fitness
Random displacements added to offspring	Mutation
Search strategy	Mutation rate, gene robustness
A set of solutions	Population

Binary representation operators

0	1	1	0	1	0	0	0

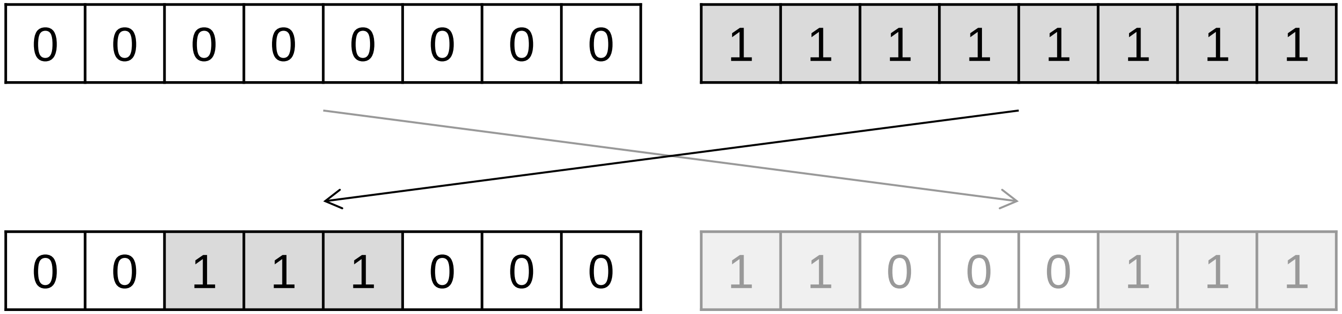
Bit flip mutation

- Each bit is inverted with a probability p_m



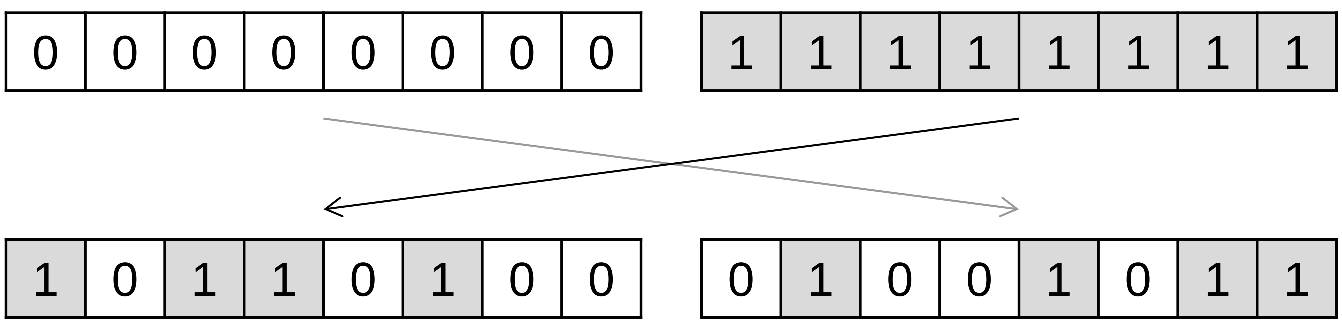
N-point crossover

- N random points in the genotype is chosen
- At each point the source parent changes



Uniform crossover

- Which parent to inherit from is chosen randomly for each position
- Identical to discrete recombination



Binary coding of integers

- Encoding integers as blocks of a binary string has been quite common
  - Keeps the analogy to DNA clean
  - Problematic because mutations are not local
    - Small changes to the solution are not more probable
    - The result of flipping a single bit varies enormously with bit position and the value of all bits that encode the same integer

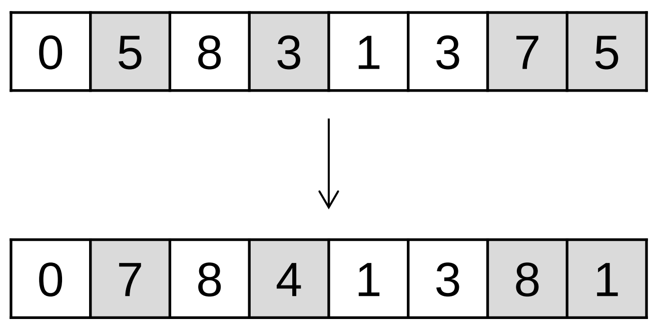
Integer representation operators

- Can use the same crossover operators as the binary representation

0	5	8	3	1	3	7	5

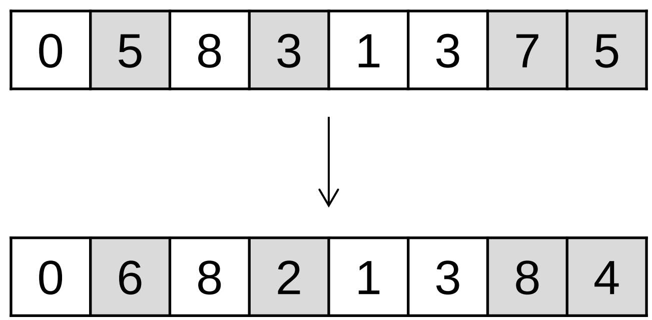
Random reset mutation

- Each element is reset with probability p_m to a random number in the range



Creep mutation

- Adds a small value to each element with probability p_m



Integer coding of symbols

- Sometimes a vector of symbols with no clear order is the most reasonable representation choice

- In such cases, the symbols are usually enumerated and treated as integers, but without using the creep mutation

Symbol	Value
N	0
E	1
S	2
W	3

Real-valued representation operators

0.1	3.3	1.7	3.4	7.2	5.9

Uniform mutation

- Each element has a probability p_m of being replaced with a number from some range

0.1	3.3	1.7	3.4	7.2	5.9

↓

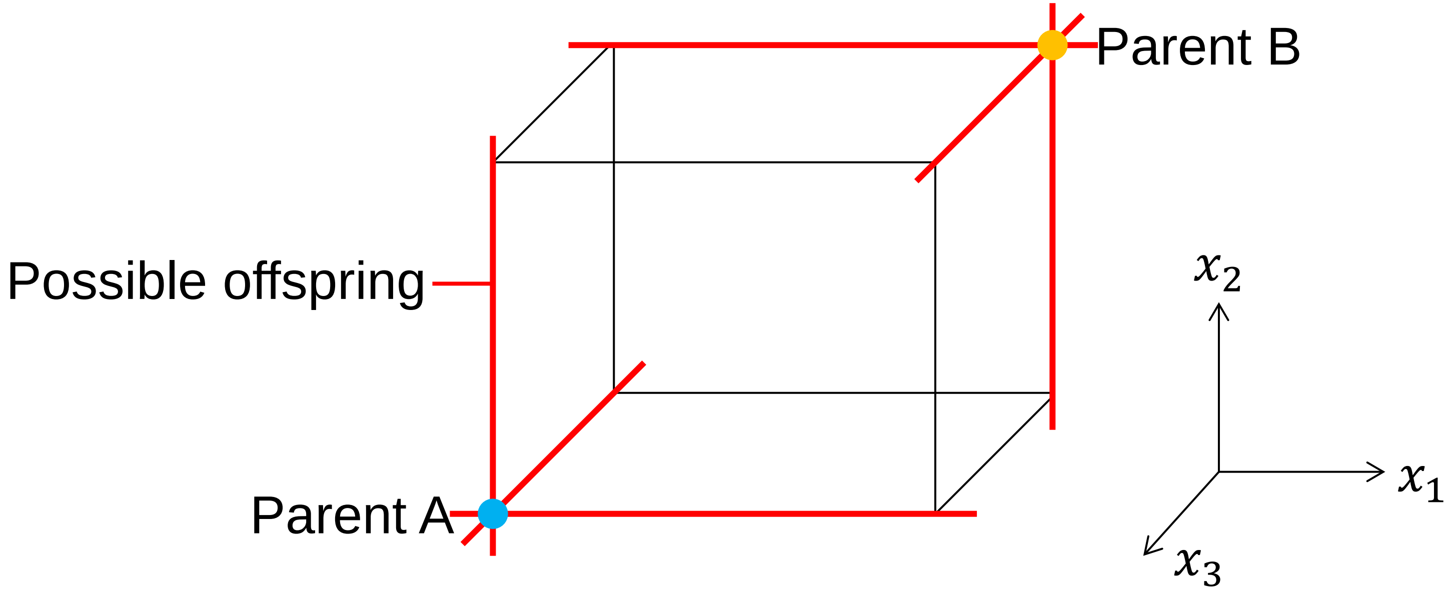
0.1	3.3	6.1	3.4	5.0	5.9

Arithmetic recombination

- Makes a copy of one of the parents x and y
- Picks one or more random positions k and replaces those elements with the interpolation $\alpha x_k + (1 - \alpha) y_k$, where α is either a fixed number or a random variable.
- Intermediate recombination: α is 0.5 for all k

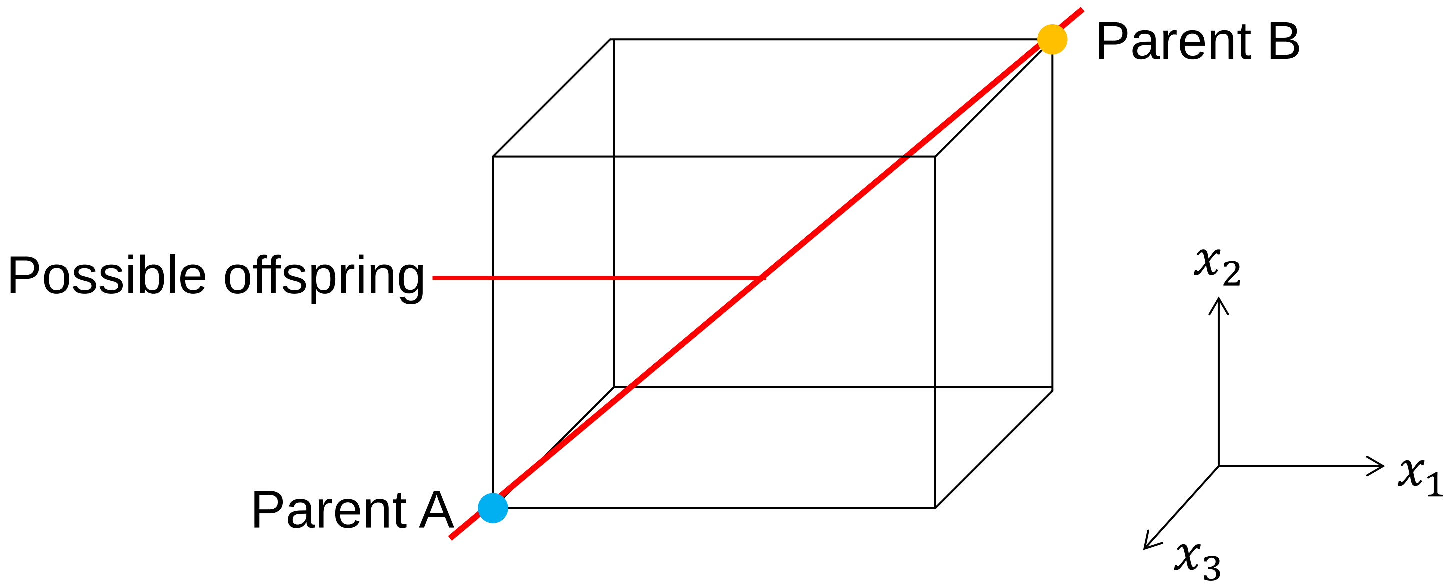
Single arithmetic recombination

- Arithmetic recombination is applied to only one k



Whole arithmetic recombination

- Arithmetic recombination is applied with the same α to all k

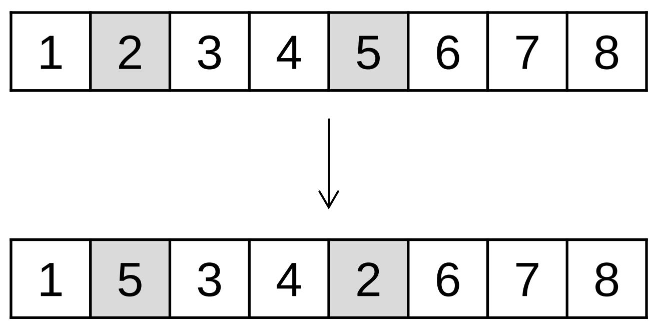


Permutation representation

- Special mutation/recombination operators
  - Each item should appear once and only once
  - Result should be “close” to the original solution(s)

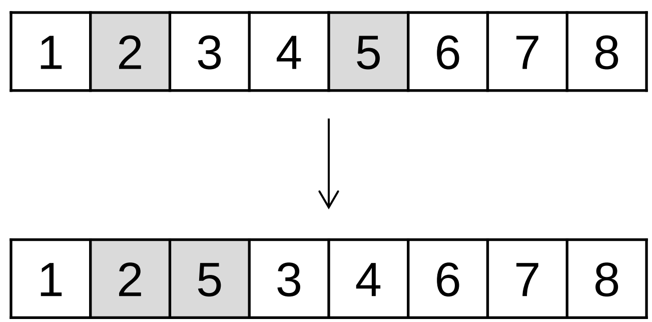
Swap Mutation

- Two random elements are swapped
- In some variants neighbors are always chosen



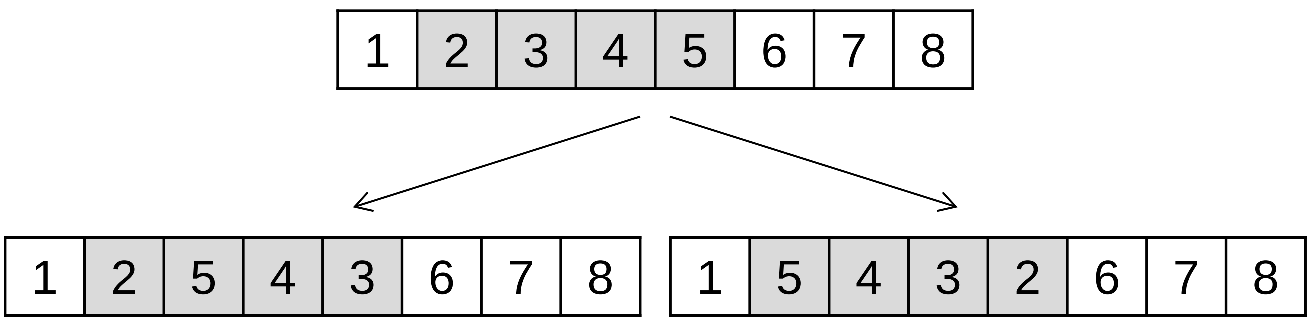
Insert mutation

- Two random elements are picked
- The second is placed right after the first



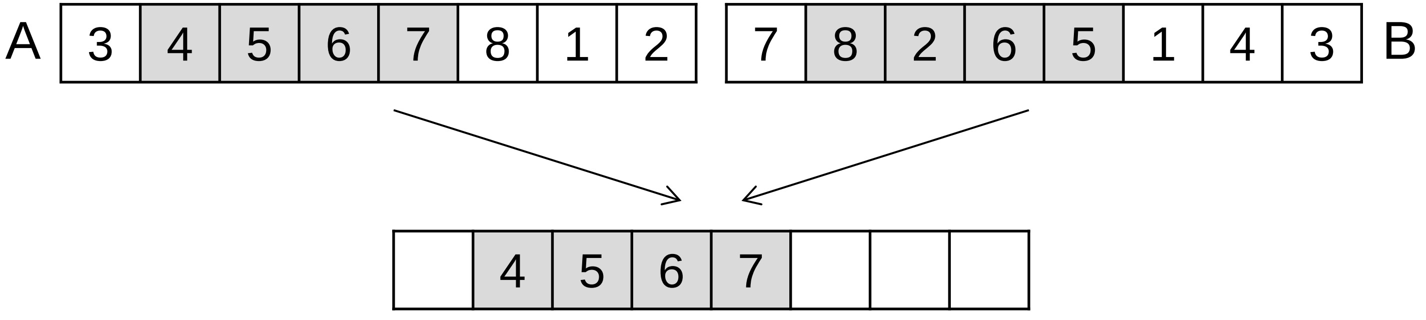
Scramble & invert mutation

- Two random points are selected
- The order of the elements in between is scrambled (scramble mutation) or reversed (invert mutation)



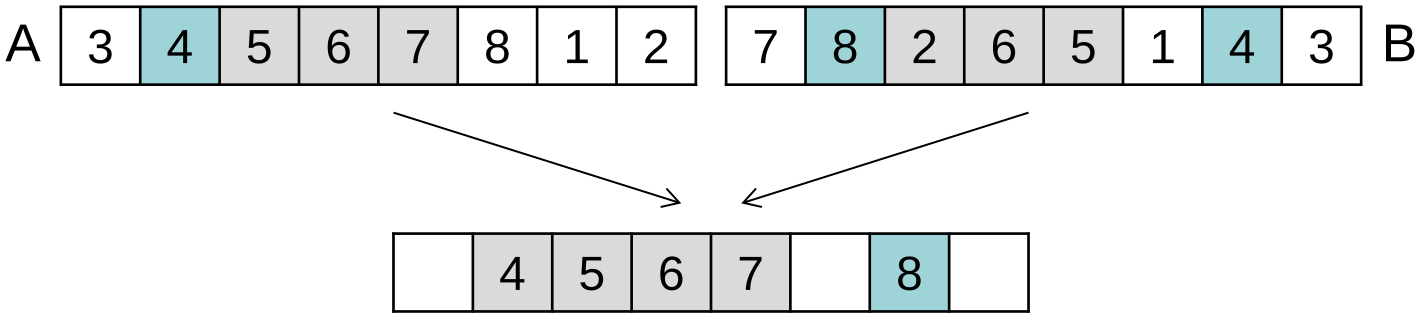
Partially mapped crossover (PMX)

- Two random points are chosen
- All elements between the points in parent A are copied to the offspring



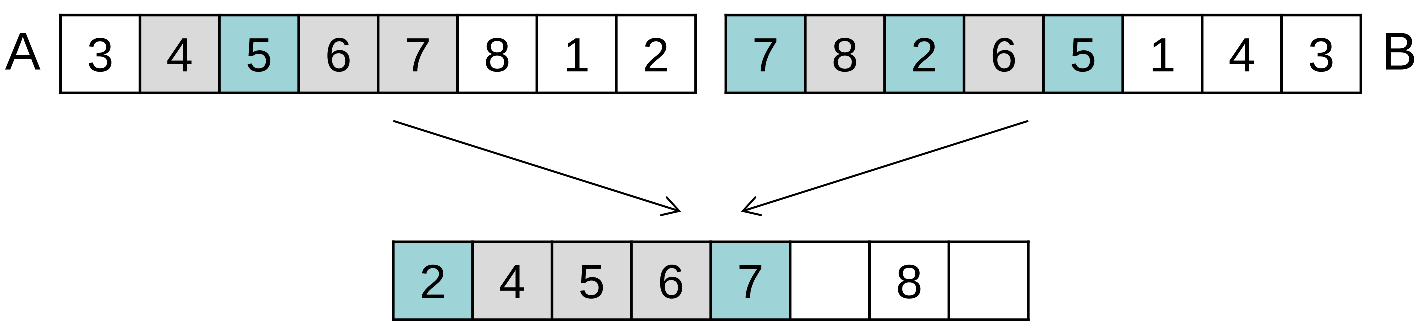
Partially mapped crossover (PMX)

- For each element x in parent B between those points that is not in parent A
  - Place it in the position in B of the element with the same position in A as x has in B



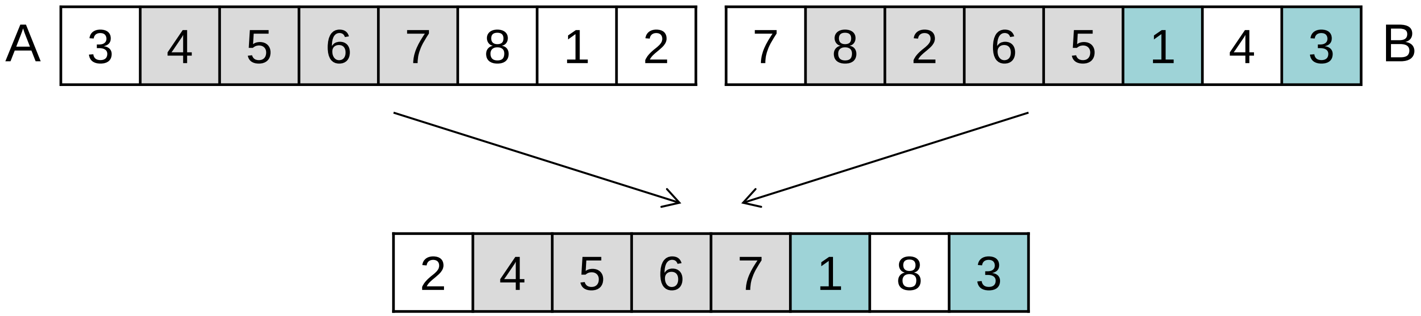
Partially mapped crossover (PMX)

- For each element x in parent B between those points that is not in parent A
  - Place it in the position in B of the element with the same position in A as x has in B
  - If that position is occupied, do one more redirection



Partially mapped crossover (PMX)

- Finally, the missing elements are copied from their places in parent B



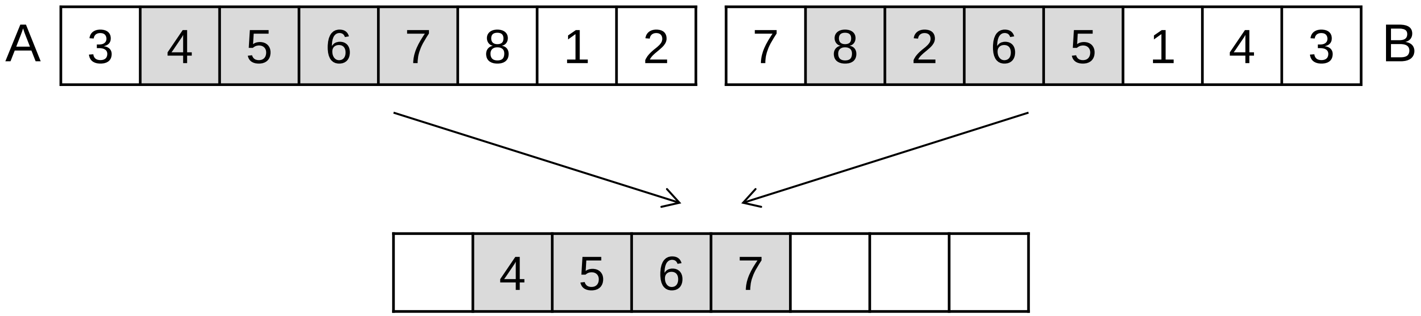
Edge crossover

- Heuristic to preserve as many edges as possible

```
def edge_xo(PA, PB, N):
    e = construct_edge_table()
    k = random(N)
    for I in range(1, N):
        X.append(k)
        e.remove(k)
        if e.empty(k): k = reverse(X)[-1]
        if e.empty(k): k = draw(1,e.remaining())
        else:
            k = e.pick_common(k) or draw(1, e.pick_shortest(k))
    return X
```

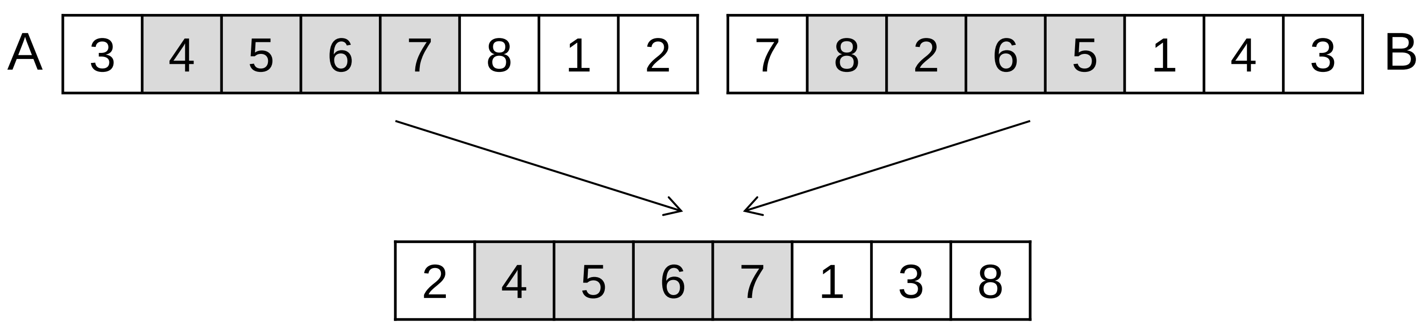
Order crossover

- Two random points are chosen
- All elements between the points in parent A are copied to the offspring



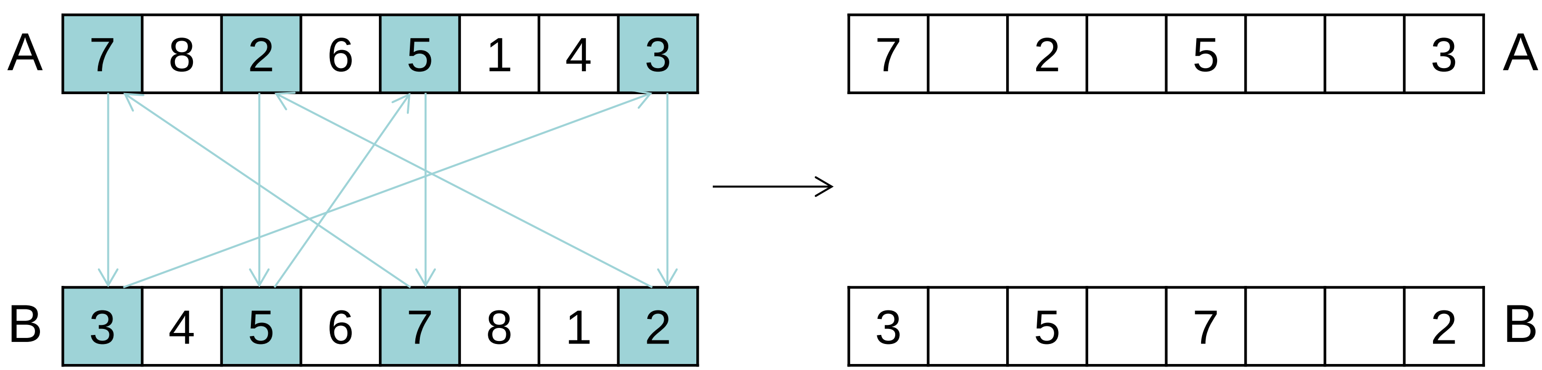
Order crossover

- The rest of the elements are copied from parent B in the order starting from the second random point



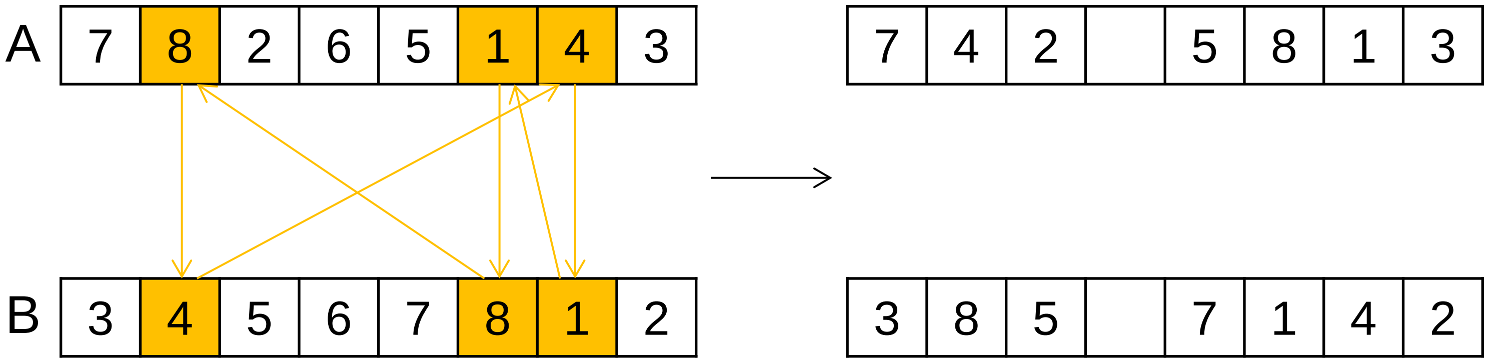
Cycle crossover

- Identify first cycle
- Copy from parent A and B to offspring A and B



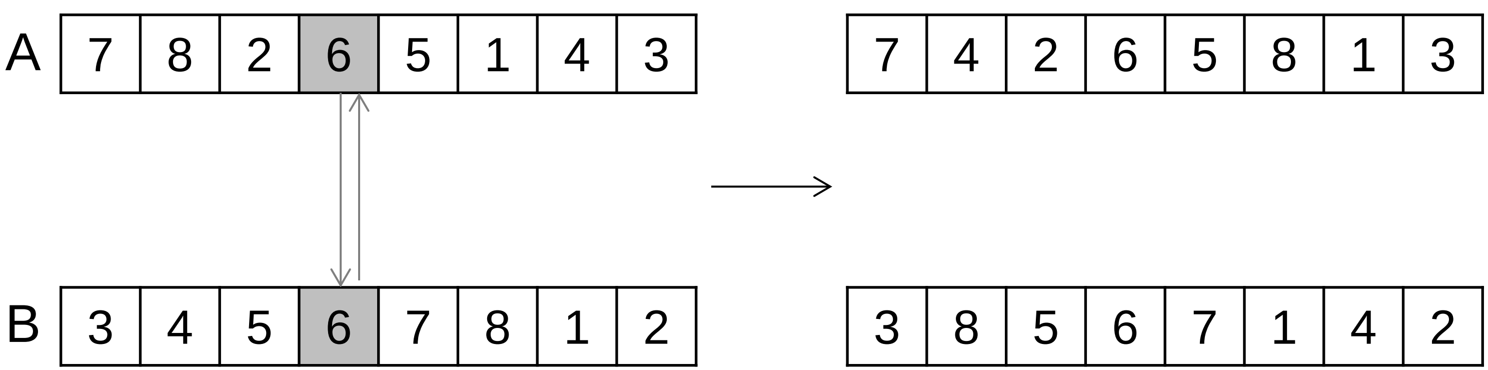
Cycle crossover

- Identify next cycle
- Copy from parent A and B to offspring B and A

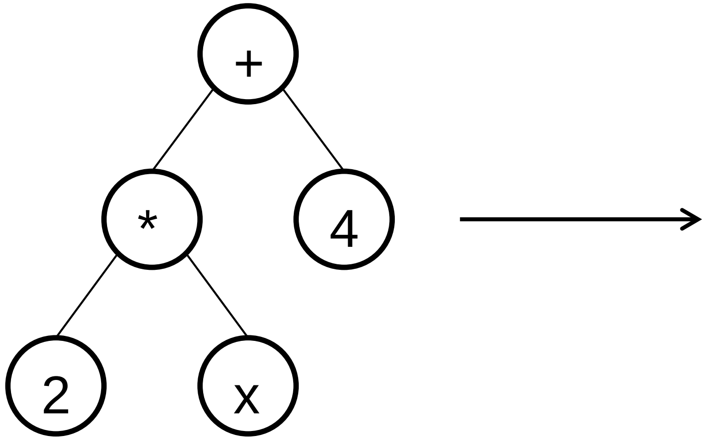


Cycle crossover

- Identify last cycle
- Copy from parent A and B to offspring A and B



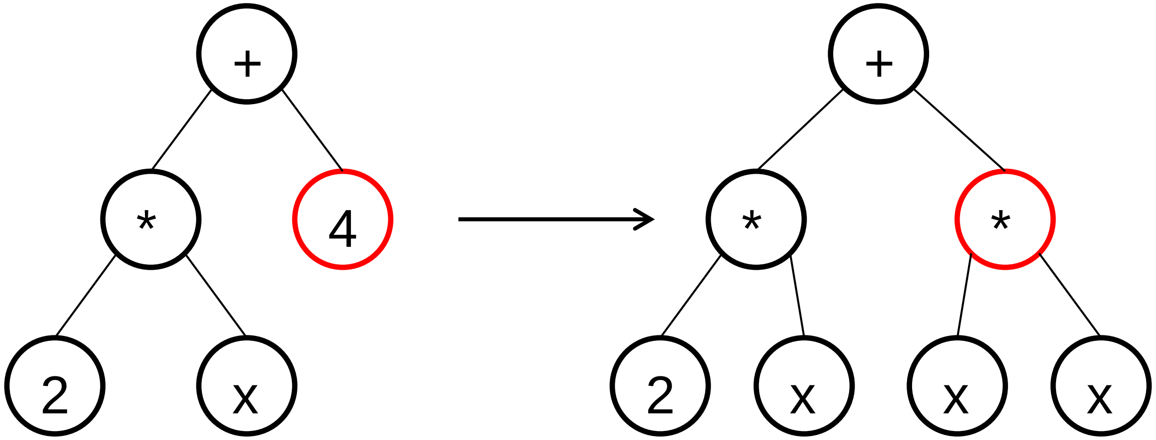
Tree representation operators



$$2x + 4$$

Tree mutation

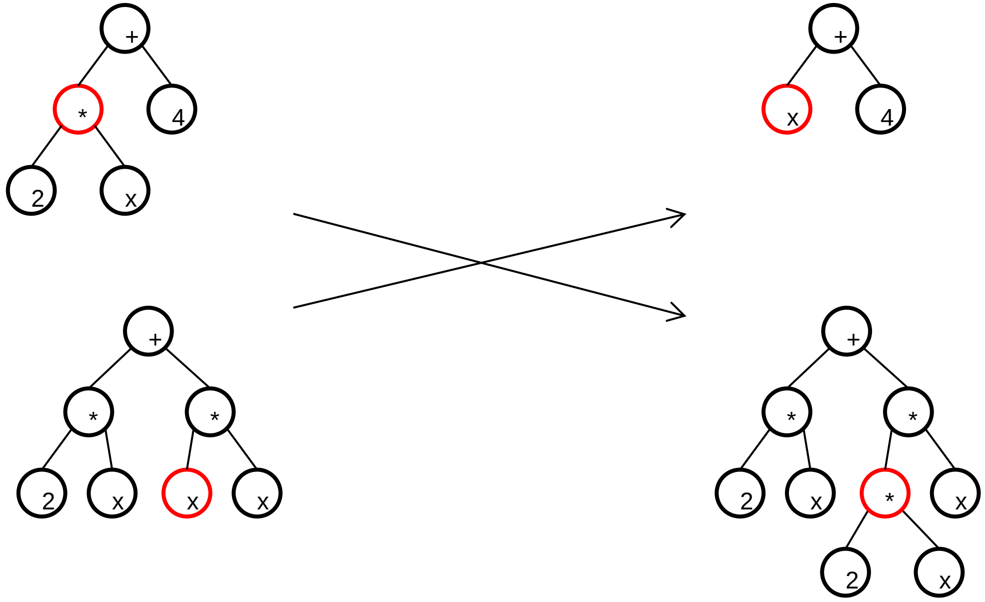
- Take a random node and replace it by a new randomly generated subtree



$2x + 4$ $\qquad$ $2x + x^2$

Tree crossover

- Take one random node from each parent and exchange them



Bloat in tree representations

- Larger trees will have greater fitness on average in most cases

- Without any active countermeasures the population will tend to grow indefinitely