Evolutionary Algorithms

Nuno Antunes Ribeiro

Assistant Professor

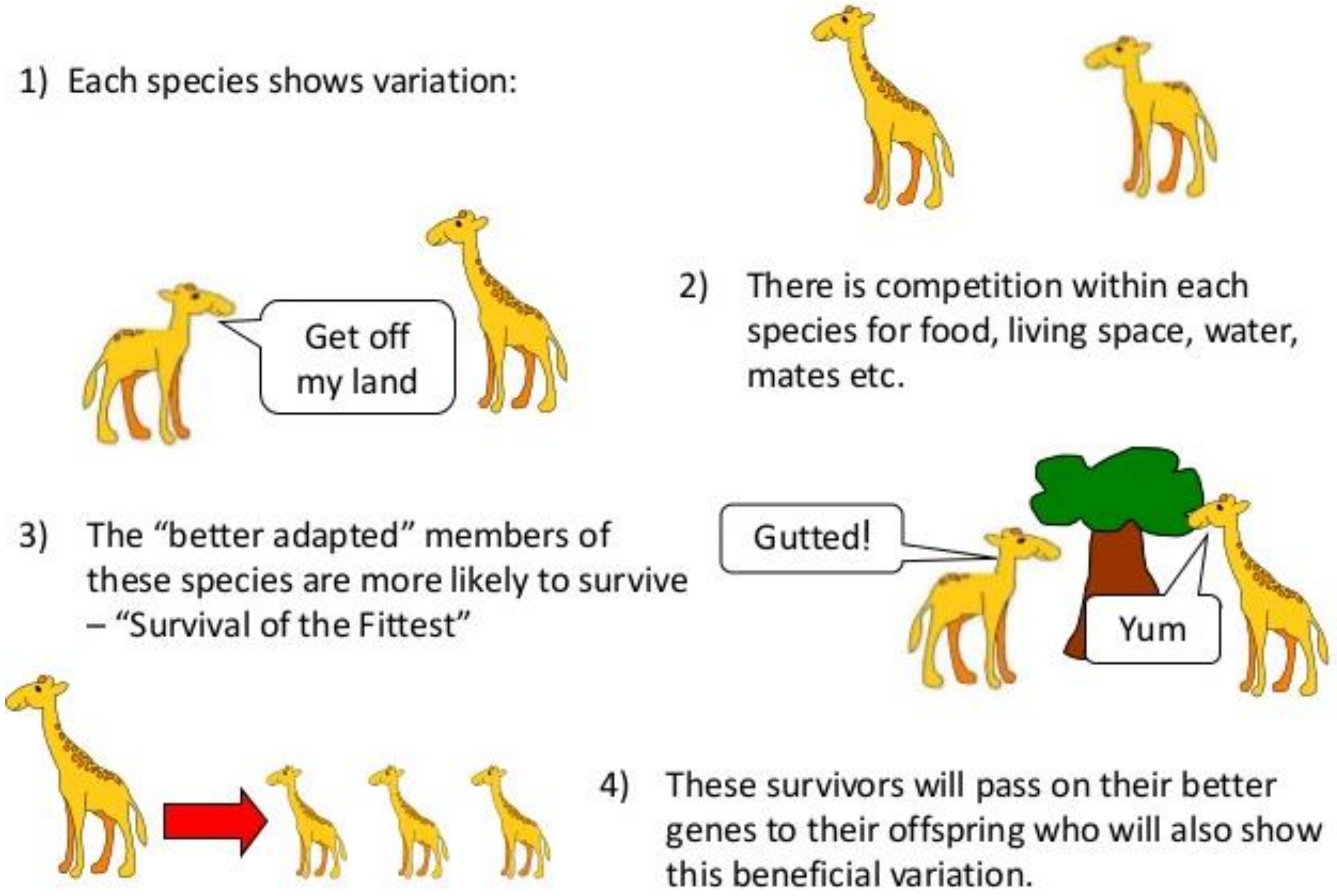
SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Engineering Systems and Design

Natural Selection

- Charles Darwin - On the Origin of Species - controversial and very influential book (1859)
  - On the origin of species by means of natural selection, or the preservation of favored races in the struggle for life
- Observations:
  - Species are continually developing
  - Homo sapiens sapiens and apes have common ancestors
  - Variations between species are enormous
  - Huge potential for production of offspring, but only a small/moderate percentage survives to adulthood

Natural Selection

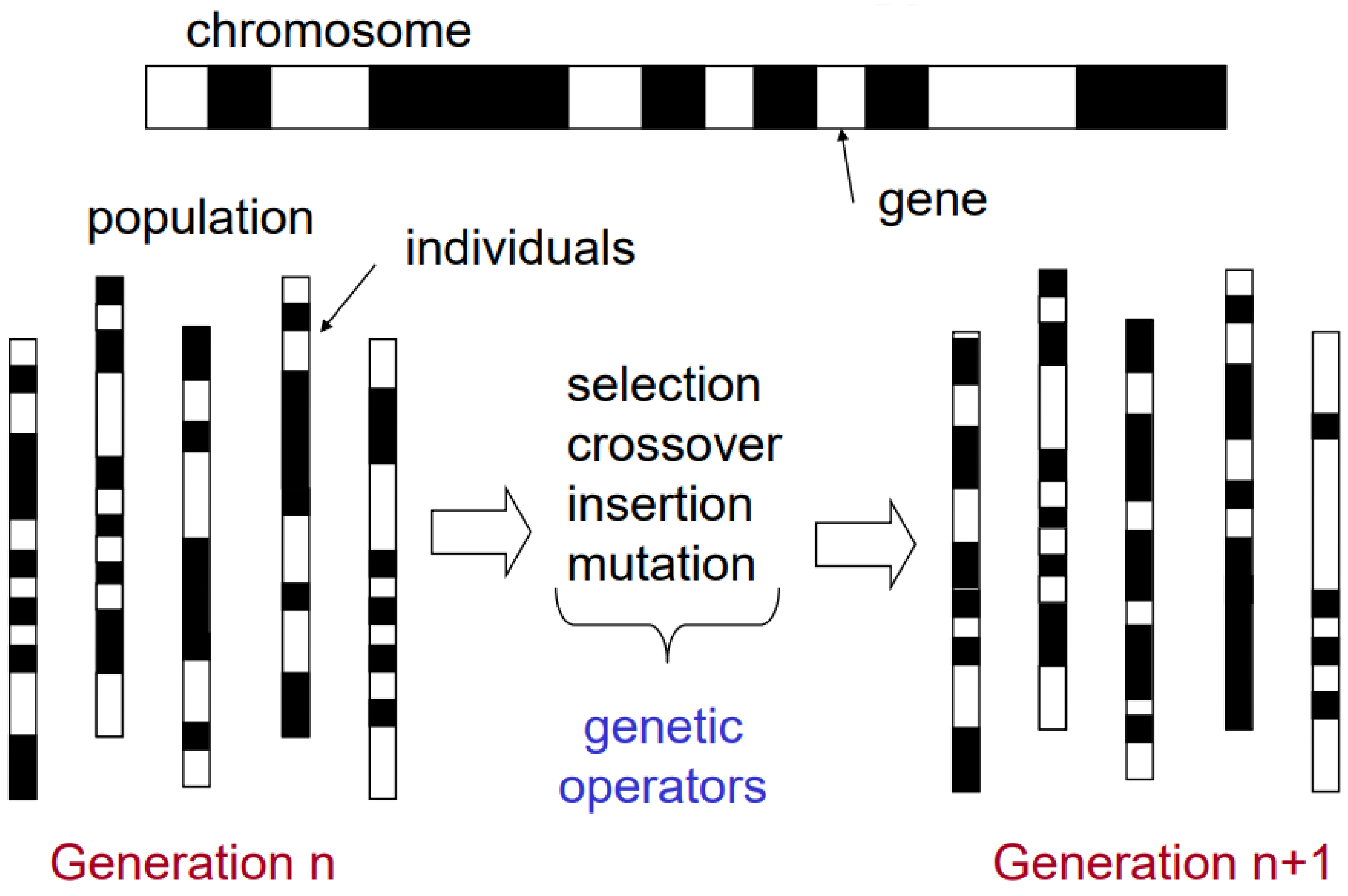


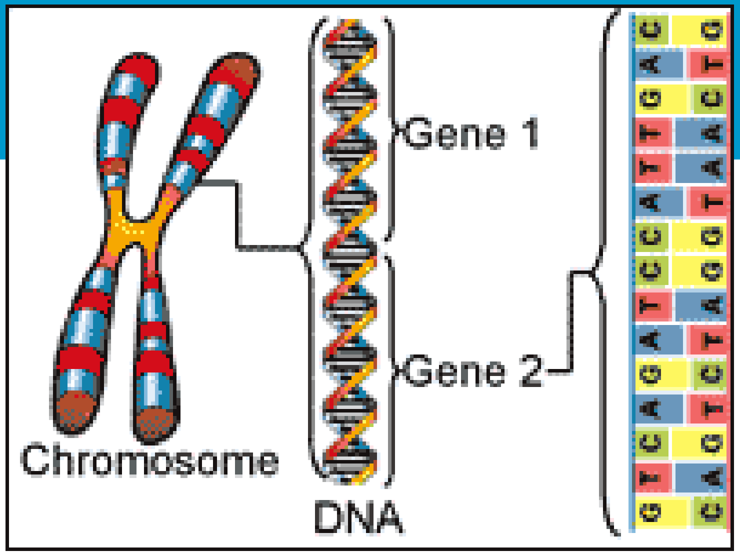
Source:http://biologyforcsec.blogspot.com/2015/12/natural-selection.html

From Natural Selection to Optimization

Metaphor	**Optimization**
Individual (*Chromosome*)	Solution
Population	Solution Space
Generations	Iterations
Fitness	Objective Value
Environment	Optimization Problem
Gene	Element of the solution
Locus	Index (position) of the solution
Allele	Possible values of each gene

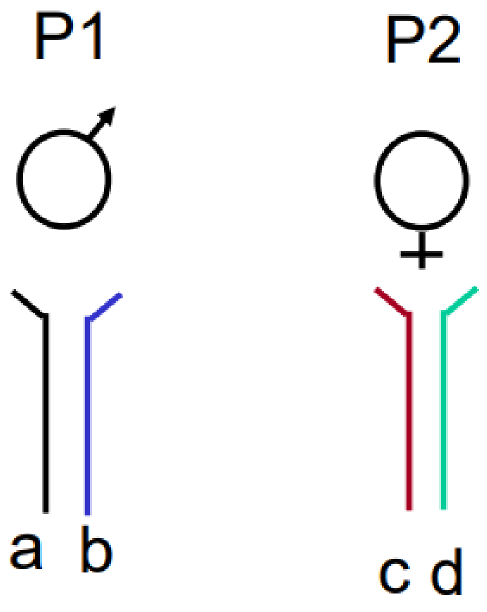
Chromosome

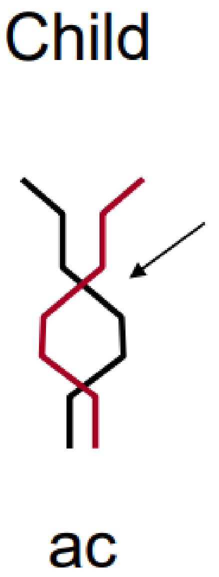




DNA is packaged in the cell into structures called chromosomes. **Chromosomes** are the form by which genetic information is passed from old cells to new cells, one generation to the next, resulting in the successful and reliable inheritance of traits. **Genes** are small segments of chromosomes.

Crossover in Biology



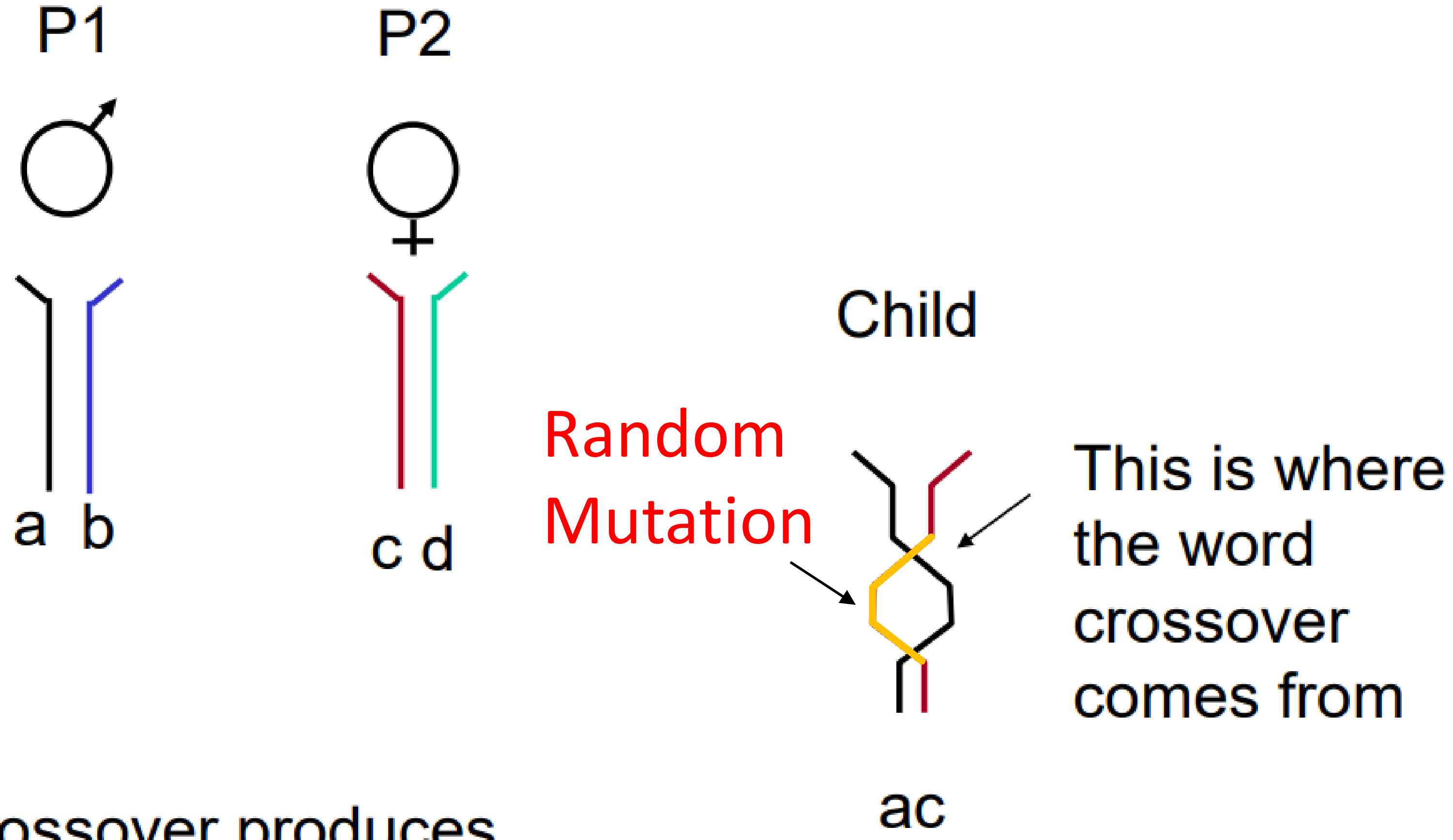


This is where the word crossover comes from

Crossover produces either of these results for each chromosome

ac OR ad OR bc OR bd

Mutation in Biology



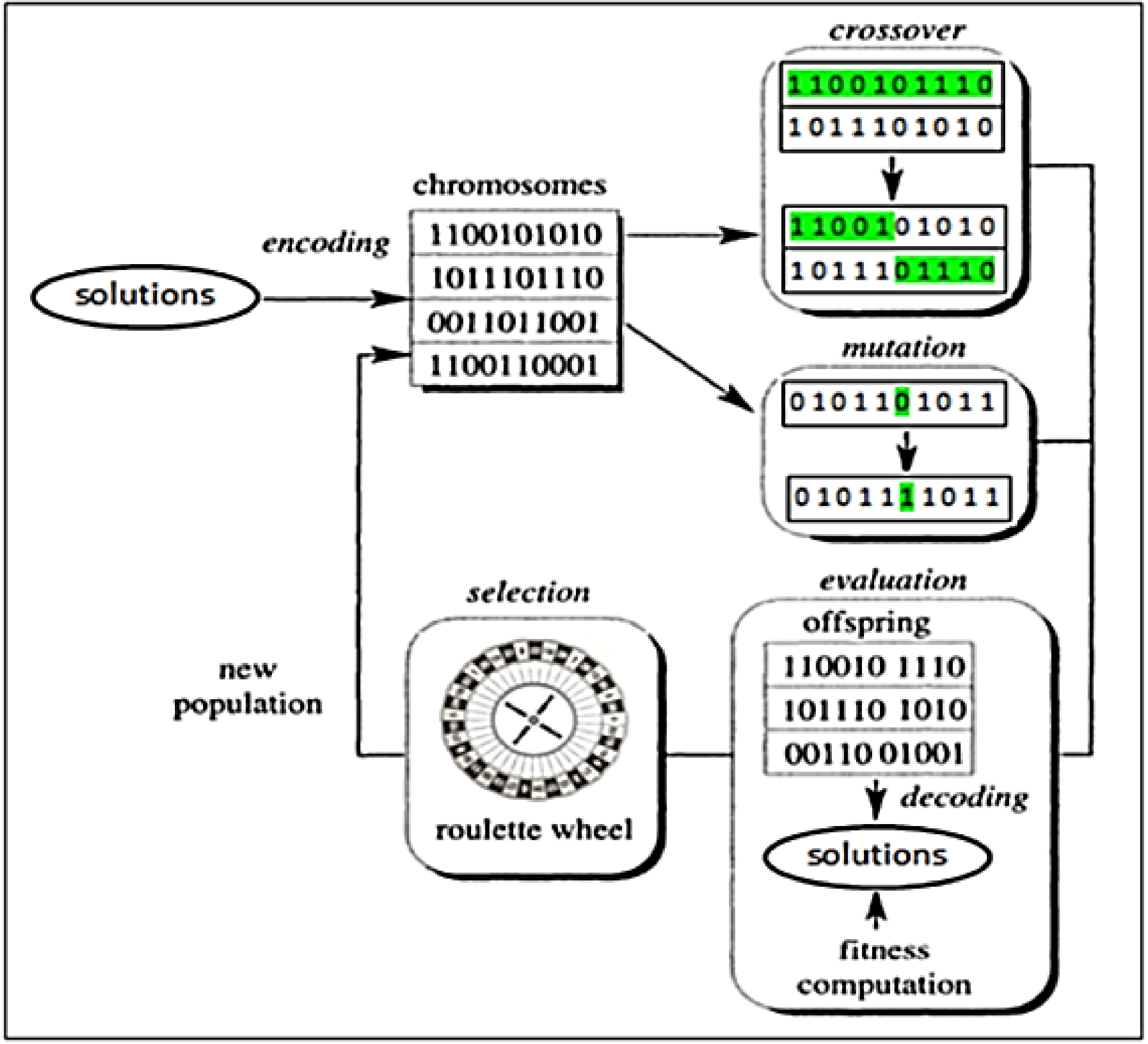
Crossover produces either of these results for each chromosome

ac OR ad OR bc OR bd

Evolutionary Algorithms (EA)

- **Evolutionary Algorithms** are based on the notion of **competition**. They represent a class of iterative optimization algorithms that **simulate the evolution of species** in a population.
- Initially, a population of **individual (chromosomes)** is generated randomly. Every individual in the population is the encoded version of a tentative solution.
- An objective function associates a **fitness value** with every individual indicating its suitability to the problem.
- At each step, individuals are selected following a **selection paradigm** in which individuals with better fitness are selected with a higher probability.
- The selected individuals are reproduced using variation operators (e.g., **crossover, mutation**) to generate new offsprings.
- Finally, a **replacement scheme** is applied to determine which individuals of the population will survive from the offsprings and the parents.
- This iteration represents **a generation** (Fig. 3.7). **This process is iterated** until a stopping criteria hold.

EA Algorithm



Traditional structure of genetic algorithm (adapted from (Gen & Cheng 1997, p.3))

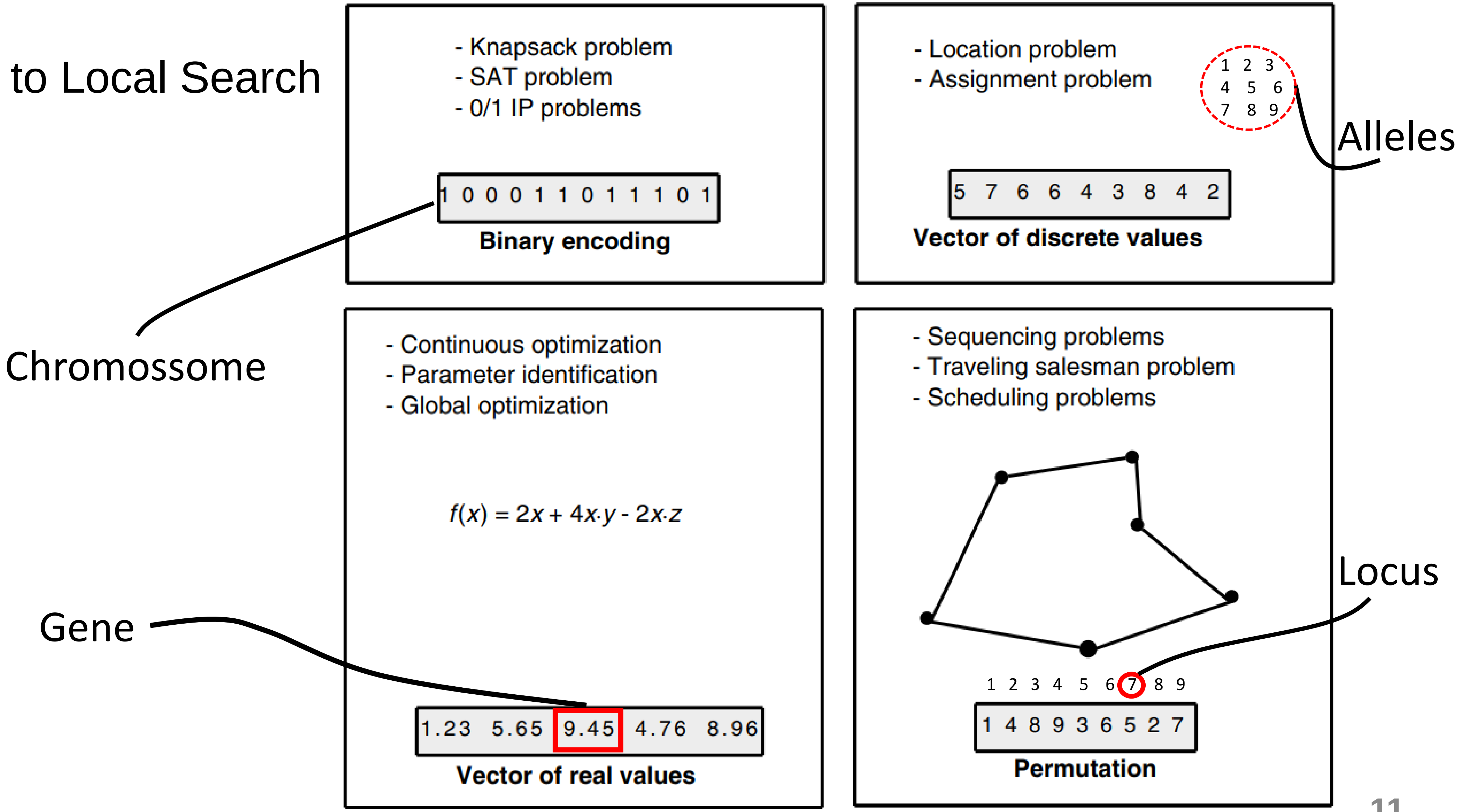
Source:https://www.youtube.com/watch?v=hDuECNoxK6s

Common Concepts in EA

- **Representation:** Similarly to local Search algorithms, solutions are encoded. The encoded solutions is referred as **chromosome**, while the decision variables within a solution are **genes**. The possible values of variables are <u>alleles</u> and the position of a decision variable within a solution is named <u>locus</u>
- **Selection Strategy:** The **selection strategy** addresses the following question: “Which parents for the next **generation** (iteration) are chosen with a bias toward better fitness?
- **Reproduction Strategy:** The reproduction strategy consists in designing suitable **mutation and crossover operator(s)** to generate new individuals (offsprings).
- **Replacement strategy:** The new offsprings **compete** with old individuals for their place in the next generation

Representation

- Similar to Local Search



Common Concepts in EA

- **Representation:** Similarly to local Search algorithms, solutions are encoded. The encoded solutions is referred as **chromosome**, while the decision variables within a solution are **genes**. The possible values of variables are alleles and the position of a decision variable within a solution is named locus
- **Selection Strategy:** The **selection strategy** addresses the following question: "Which parents for the next **generation** (iteration) are chosen with a bias toward better fitness?
- **Reproduction Strategy:** The reproduction strategy consists in designing suitable **mutation and crossover operator(s)** to generate new individuals (offsprings).
- **Replacement strategy:** The new offsprings **compete** with old individuals for their place in the next generation

Selection Strategy

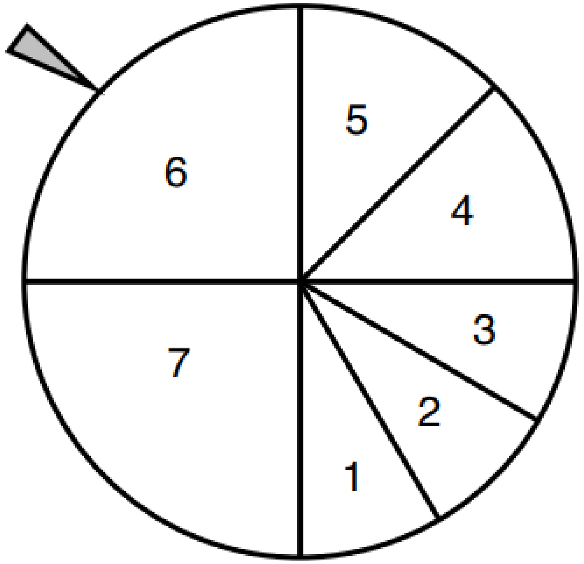
- The main principle of selection methods is "**the better is an individual, the higher is its chance of being parent**." Such a selection pressure will drive the population to better solutions.
- However, **worst individuals** should not be discarded and they have some chance to be selected. This may lead to **useful genetic material**.
- Typically, the parents are selected according to their fitness by means of one of the following strategies:
  - roulette wheel selection,
  - stochastic universal sampling (SUS),
  - rank-based selection,
  - tournament selection.

Roulette Wheel Selection

- It will assign to each individual a **selection probability** that is proportional to it relative fitness. Let f_i be the fitness of the individual p_i in the population P. Its probability to be selected is
$$p_i = f_i \Big/ \left(\sum_{j=1}^{n} f_j\right)$$

- Suppose a **pie graph** where each individual is assigned a space on the graph that is **proportional to its fitness**. An outer roulette wheel is placed around the pie. The selection of μ individuals is performed by μ independent spins of the roulette wheel.

Individuals:	1	2	3	4	5	6	7
Fitness:	1	1	1	1.5	1.5	3	3

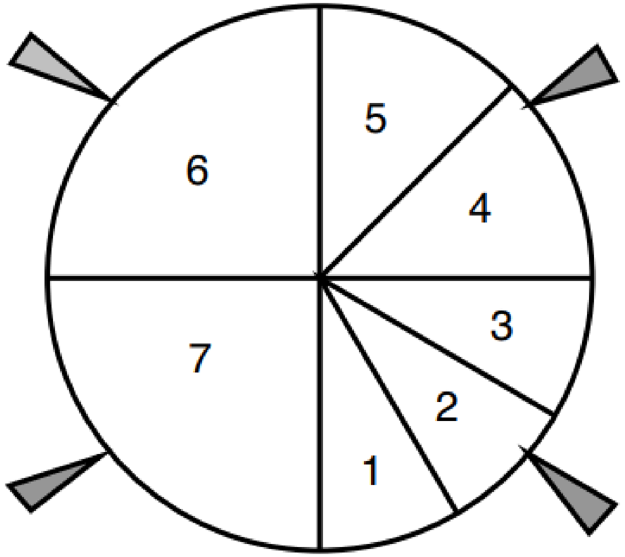


Roulette selection

Stochastic Universal Sampling (SUS)

- In the roulette wheel selection, **outstanding individuals will introduce a bias** in the beginning of the search that may cause a premature convergence and a loss of diversity.
- To reduce the bias of the roulette selection strategy, an outer roulette wheel is placed around the **pie with μ equally spaced pointers**. A single spin of the roulette wheel will simultaneously select μ individuals for reproduction.

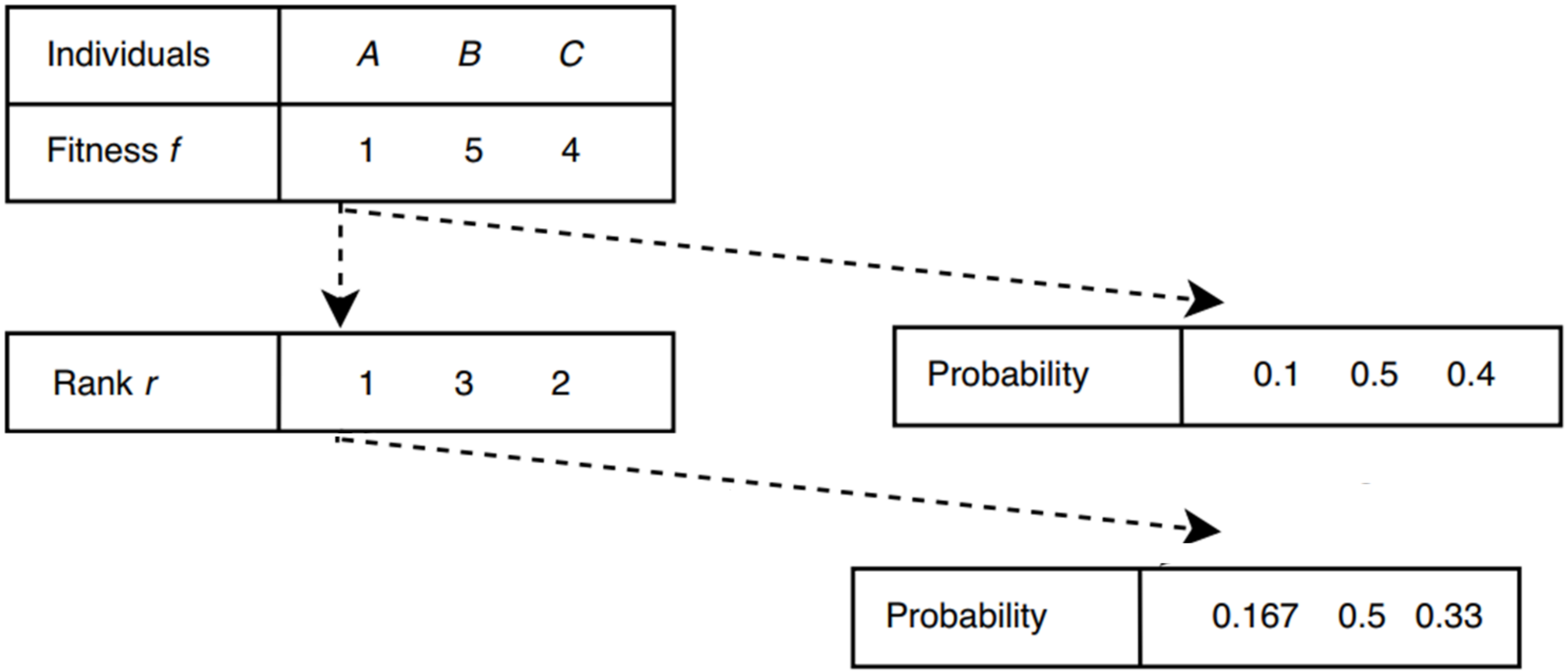
Individuals:	1	2	3	4	5	6	7
Fitness:	1	1	1	1.5	1.5	3	3



Stochastic universal sampling

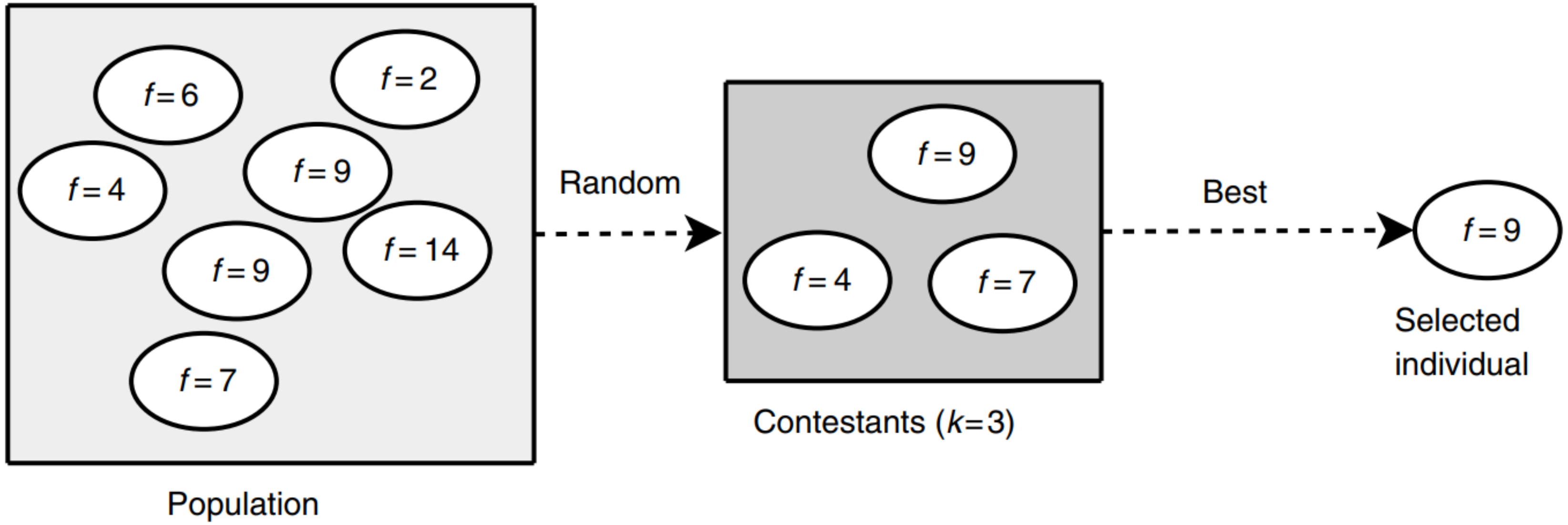
Rank-Based Selection

- Instead of using the fitness value of an individual, the **rank of individuals** is used. The function is biased toward individuals with a high rank.
- The rank may be scaled linearly using the following formula:



Tournament Selection

- Tournament selection consists in **randomly selecting k individuals**; the $parameter\ k$ is called the size of the tournament group. A tournament is then applied to the k members of the group to select the best one.
- To select μ individuals, the **tournament procedure is then carried out μ times**.



Tournament selection strategy. For instance, a tournament of size 3 is performed. Three solutions are picked randomly from the population. The best solution from the picked individuals is then selected.

Common Concepts in EA

- **Representation:** Similarly to local Search algorithms, solutions are encoded. The encoded solutions is referred as **chromosome**, while the decision variables within a solution are **genes**. The possible values of variables are alleles and the position of a decision variable within a solution is named locus
- **Selection Strategy:** The **selection strategy** addresses the following question: “Which parents for the next **generation** (iteration) are chosen with a bias toward better fitness?
- **Reproduction Strategy:** The reproduction strategy consists in designing suitable **mutation and crossover operator(s)** to generate new individuals (offsprings).
- **Replacement strategy:** The new offsprings **compete** with old individuals for their place in the next generation

Crossover

- The role of crossover operators is to inherit some characteristics of the **two parents to generate the offspring's** (similar to local search, we aim to take advantage of good features already found).
- **Idea:**
  - two individuals have been selected (as parents)
  - thus, we can assume that both have good features
  - if the population is **diverse**, then the two selected individuals probably have different features
- **Goal:**
  - **Combine these different (good) features**. . .
  - . . . and obtain a new, **possible better candidate solution**
- The mutation and crossover operators are complementary.

Mutation introduces exploration; Crossover introduces exploitation.

Crossover Operators

- There exist different crossover operators. As for the mutation operator, the design of crossover operators mainly depends on the representation used
- **Discrete & Binary Operators**

Single-Point Crossover (SPX)

Parents

5	4	4	2	5	2	1	1	4	5
3	3	3	2	5	3	3	1	5	6

Offsprings

5	4	4	2	5	3	3	1	5	6
3	3	3	2	5	2	1	1	4	5

Parents

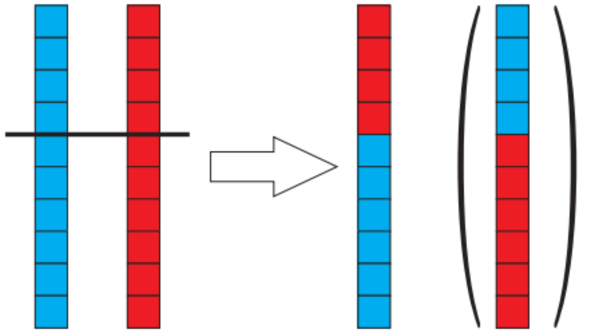
1	0	0	0	1	1	1	0	0	1
0	0	1	1	1	1	0	1	0	0

Offsprings

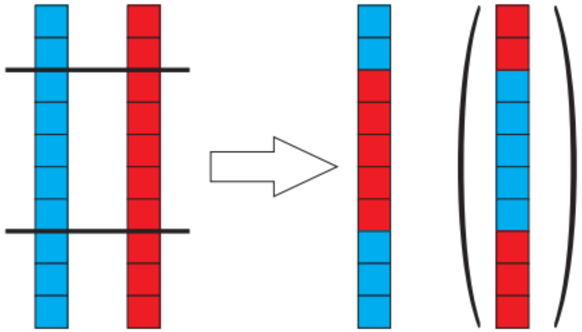
1	0	0	0	1	1	0	1	0	0
0	0	1	1	1	1	1	0	0	1

Crossover Operators

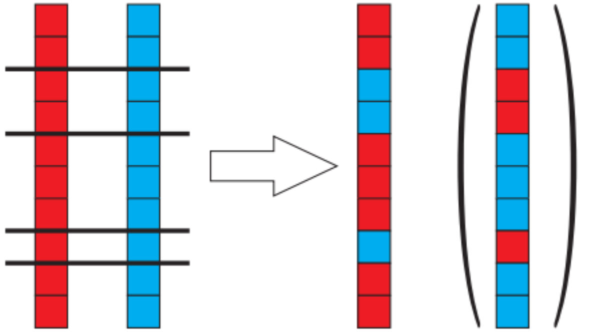
- **Discrete and Binary Operators**



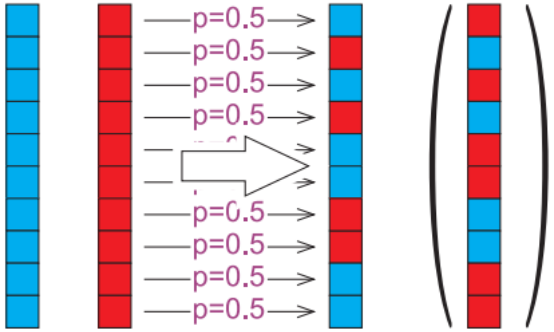
Single-Point (SPX)



Two-Point (TPX)



Multi-Point (MPX)



Uniform (UX)

Crossover Operators

- **Real Operators**

Parents

0.10	0.23	0.41	0.13	0.46	0.21	0.66	0.22	0.19	0.83	**P1**
0.15	0.31	0.22	0.64	0.34	0.24	0.57	0.14	0.33	0.95	**P2**

$\alpha P1_i + (1-\alpha)P2_i \longrightarrow \alpha$ generated randomly U(0,1)

Offspring

0.14	0.29	0.28	0.49	0.38	0.23	0.60	0.16	0.29	0.91	e.g. $\alpha = 0.3$

Crossover Operators

- **Permutation Operators**
- Applying classical crossover operators to permutations will generate solutions that are not permutations (i.e., infeasible solutions).

Parents

1	4	3	7	5	9	6	2	8	10

3	7	10	8	1	4	6	2	5	9

Offsprings

1	4	3	7	1	4	6	2	5	9

3	7	10	8	5	9	6	2	8	10

Infeasible solutions!!

Crossover Operators

- **Permutation Operators**
- Order Crossover (OX):

Parents

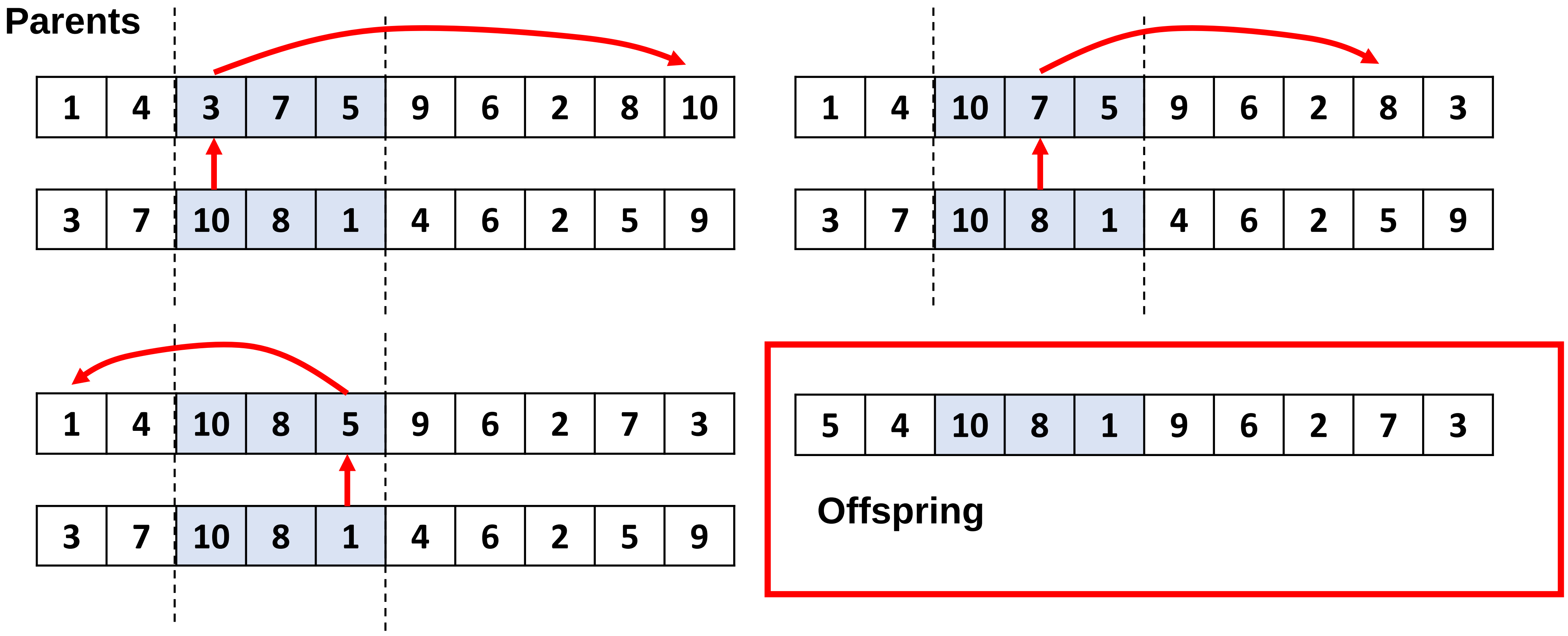
1	4	3	7	5	9	6	2	8	10

3	7	10	8	1	4	6	2	5	9

10	8	3	7	5	9	1	4	6	2

Crossover Operators

- **Permutation Operators**
- Partially Mapped Crossover (PMX):



Crossover Operators

- **Permutation Operators**
- Uniform Crossover (PMX):

Parents

1	4	3	7	5	9	6	2	8	10
0	0	1	0	1	1	1	0	0	0
3	7	10	8	1	4	6	2	5	9

Randomly generated binary vector

Offspring

7	10	3	8	5	9	6	1	4	2

Mutation Operators

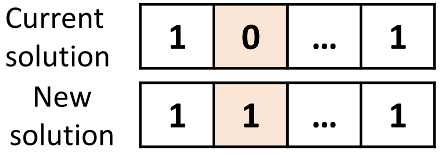
- Mutations represent **small changes** of selected individuals of the population.
- The probability p_m defines the **probability to mutate** each element (**gene**) of the representation. It can also affect only one gene too.
- In general, small values are recommended for this probability (e.g. 0.1%).
- Some strategies initialize the mutation probability to $1/k$ where k is the number of decision variables, that is, in average only one variable is mutated.

- The mutation in evolutionary algorithms is related to **neighbourhood operators of Local Search**. Hence, the neighbourhood structure definitions for traditional representations may be reused as mutation operators

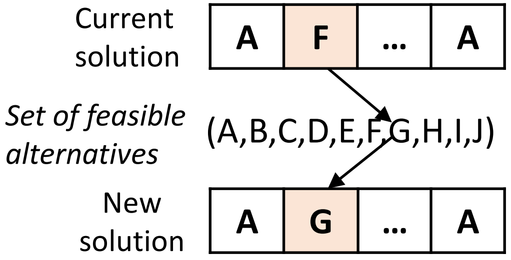
Recall: Search Operator

- The efficiency of a solution encoding is also related to the **search operator**.
- When defining a solution encoding, one has to bear in mind how the solution will be **perturbed**.
- More drastic perturbations (for instance flipping 2 bits instead of 1) encourage diversification

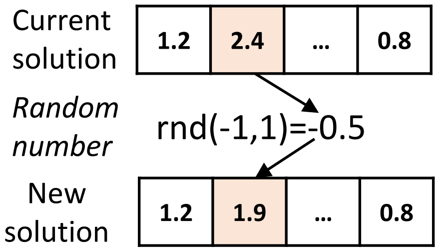
Binary encoding – flip **n** bits of the solution (typically 1 or 2 bits)



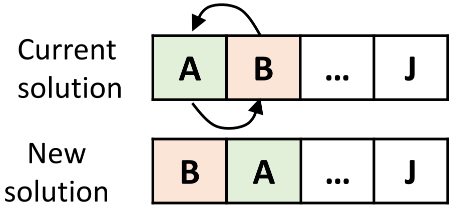
Discrete encoding – update **n** bits of the solution by randomly generating a new value (typically 1 or 2 bits)



Real encoding – update **n** bits of the solution by randomly generating a new value within a certain range (typically 2 elements)



Permutation encoding – swap the location of **n** elements (typically 2 elements)



Common Concepts in EA

- **Representation:** Similarly to local Search algorithms, solutions are encoded. The encoded solutions is referred as **chromosome**, while the decision variables within a solution are **genes**. The possible values of variables are <u>alleles</u> and the position of a decision variable within a solution is named <u>locus</u>
- **Selection Strategy:** The **selection strategy** addresses the following question: “Which parents for the next **generation** (iteration) are chosen with a bias toward better fitness?
- **Reproduction Strategy:** The reproduction strategy consists in designing suitable **mutation and crossover operator(s)** to generate new individuals (offsprings).
- **Replacement strategy:** The new offsprings **compete** with old individuals for their place in the next generation

Replacement Strategy

- The replacement phase concerns the survivor selection of both the parent and the offspring populations. As the size of the population is constant, it allows to withdraw individuals according to a given selection strategy.
- First, let us present the extreme replacement strategies:
  - **Generational replacement:** The offspring population will replace systematically the parent population.
  - **Steady-state replacement:** At each generation of an EA, only one offspring is generated. It replaces the worst individual of the parent population.

- Between those two extreme replacement strategies, many distinct schemes that consist in **replacing a given number of λ individuals** of the population may be applied ($1 < \lambda < \mu$).

Replacement Strategy

- **Elitism** always consists in selecting the best individuals from the parents and the offsprings. This approach may lead to a faster convergence and a **premature convergence** could occur.
- Elitism is usually **paired with random selection** or any of the other techniques to get a mixture of good diversity and convergence.

- **Hall of Fame** is another mechanism where the best individual of the population is set aside in the Hall of Fame. By doing this, it allows the algorithm to favour good exploration with random selection without the fear of losing the best individual – the difference to elitism is that the best individual of each generation is kept, even if much better solutions have been found in subsequent iterations

Common parameters of EAs

- **Crossover probability $\boldsymbol{p_c}$:** The crossover probability is generally set from medium to large values (e.g., $p_c \in [0.3, 1]$).
- **Mutation probability $\boldsymbol{p_m}$:** A large mutation probability will disrupt a given individual and the search is more likely random. Generally, small values are recommended for the mutation probability ($p_m \in [0.001, 0.01]$). Usually, the mutation probability is initialized to $1/k$ where k is the number of decision variables. Hence, on average, only one variable is mutated.
- **Population size $\boldsymbol{\mu}$:** The larger is the size of the population, the better is the convergence toward "good" solutions. However, the time complexity of EAs grows linearly with the size of the population. A compromise must be found between the quality of the obtained solutions and the search time of the algorithms. In practice, the population size is usually between 20 and 100.

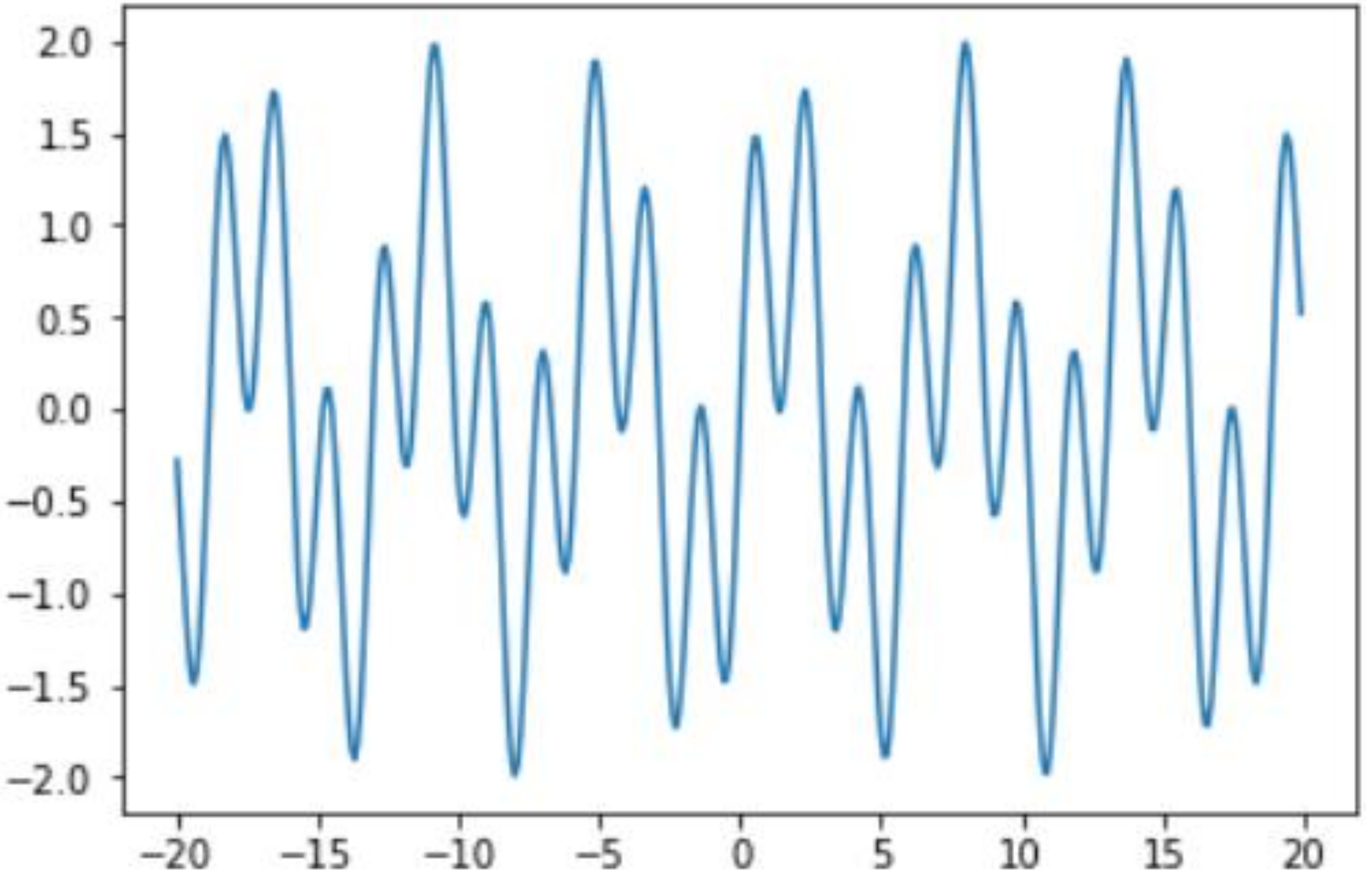
Initial Population

- The **Initial population** of EC algorithms are extremely important. Evolutionary Algorithms (EA) are population based search algorithms, meaning it works by taking a pool of initial points and searches these points in parallel.
- Unlike local search methods where you only feed it one initial value, EA's work by using the **diversity** of the population to search for better solutions.
- If our initial population only includes values from a centrally located part of the search space then our algorithm will only find optimal solutions near that search space (mutation operators may help to solve this issue)
- To ensure diversity, we need to **randomly sample uniformly** from our domain space.
- Whenever we are dealing with **constrained problems**, our initial uniform sampling might lead to infeasible solutions. To get around this we either have to hard code these constraints, or just sample over the entire input space but only keep feasible solutions.

Basic Example

- Suppose we have the following function below we want to maximize:

$$f(x) = sin(x) + \sin(3.33x)$$



EA Design Inputs

- **Initial Population:** uniformly randomly generated from the domain space of [-20, 20]
- **Population Size:** Because this is such a trivial example we could have an initial size of thousands as the fitness function would not take long to evaluate; but, to keep things simple we will only use 5.
- **Solution Representation:** vector of real values
- **Selection Strategy:** roulette wheel selection
- **Reproduction Strategy:**

  $\alpha P1_i + (1 - \alpha) P2_i$

  - For the crossover**,** we will use the average of the parents where α is uniformly distributed from 0 to 1.
  - The mutation value will be 1% of the entire bound (i.e. 0.4)
  - Because we do not always want to perform crossover or mutation as it might lose information from the parents, we will probabilistically do so only performing these operations 75% of the time.
- **Replacement strategy :** we set elitism at 20%, meaning the best 20% of the parents will always be carried over; however, because our initial size is only 5, 20% really only means one individual.

Initial Population & Fitness Function

Initial Population

```
np.random.seed(3)

def fitness_function(x):
    return sin(x) + sin((3.333) * x)   # our fitness function is the function itself

def scale_fitness_1(x):  # scale to make all positive
    return x + np.abs(np.min(x))

def scale_fitness_2(x):  # scale for minimization
    return 1 / (1 + x)

size = 5  # initial population size
# domain
lower_bound = -20
upper_bound = 20
# initial generation -- needs to be random across entire domain for
# accurate representation
init_gen = np.random.uniform(lower_bound, upper_bound, size)
next_gen=init_gen #initial generation is the next generation to be analyzed
next_gen
```

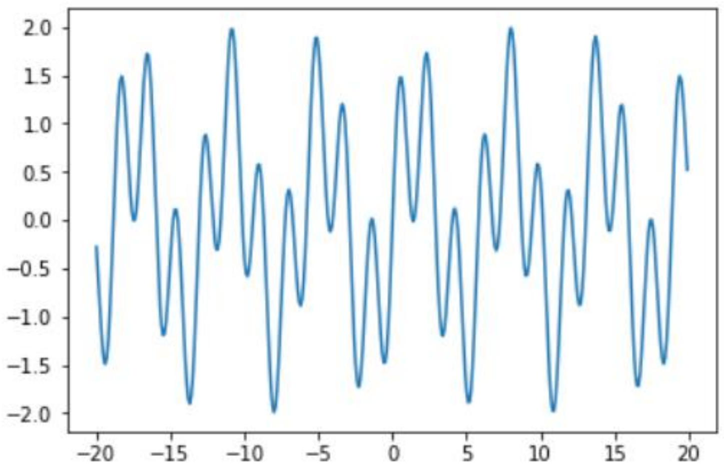
x=
```
array([ 2.0319161 ,  8.3259129 , -8.36381044,  0.43310421, 15.71787817])
```

```
fitness  # fitness values of initial generation
```

fitness=
```
array([ 1.3654668 ,  1.3910967 , -1.26017642,  1.41160378,  0.84187238])
```

- Because we are using roulette wheel selection, we cannot sum up negative fitness values;
- Therefore, we need to scale so that all fitness values are positive;
- This is done by adding the absolute value of the minimum to each value;

- If our objective is to minimize a given fitness function, then we need to scale also for minimization



Fitness Function

Compute Fitness

```
next_gen
```

x= array([2.0319161 , 8.3259129 , -8.36381044, 0.43310421, 15.71787817])

Best Solution found, it will be used later for **elitism**

```
prev_gen=np.copy(next_gen) #generate copy of current generation, it will be used later for elitism
fitness = fitness_function(next_gen)
fitness  # fitness values of initial generation
```

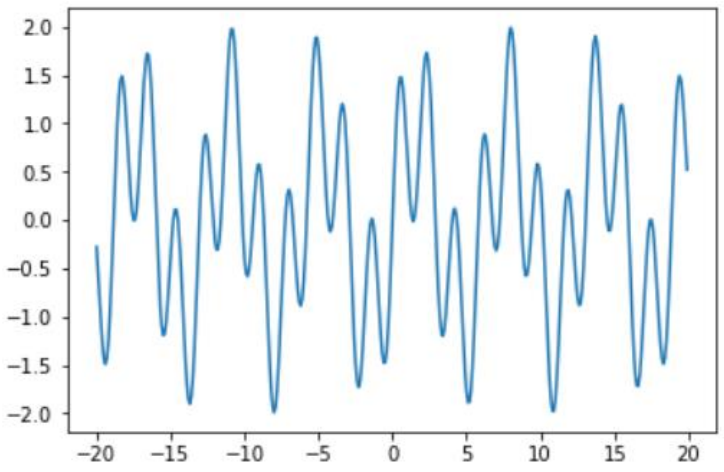
fitness= array([1.3654668 , 1.3910967 , -1.26017642, 1.41160378, 0.84187238])

```
# because we are using roulette wheel selection, we cannot sum up negative
# fitness values; therefore, we need to scale so that all fitness values are
# positive; this is done by adding the absolute value of the minimum to each
# value; here is our new scaled fitness

scaled_fit=scale_fitness_1(fitness) # scale to make all positive
scaled_fit
```

array([2.62564322, 2.65127313, 0.   , 2.6717802 , 2.1020488])

```
#If we are maximizing, we don't perform this second layer of scaling,
#but if we are then we scale using the equation below.

scaled_fit=scale_fitness_2(scaled_fit)  # scale for minimization
scaled_fit
```

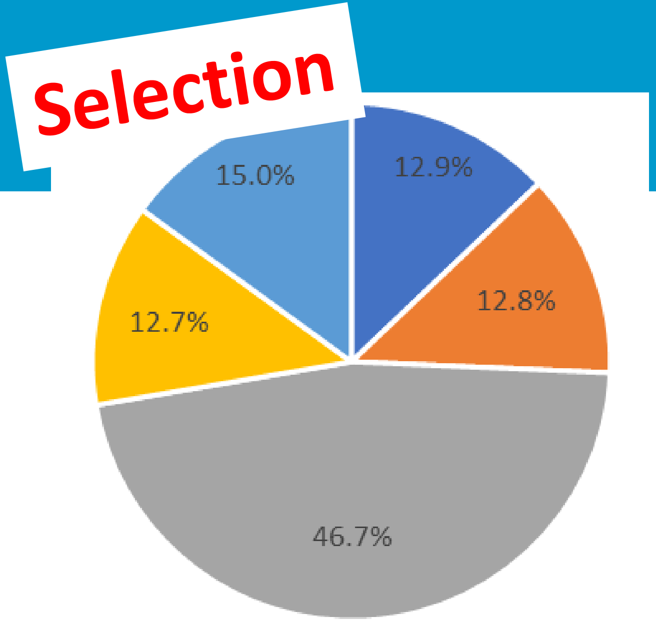
array([0.27581313, 0.27387707, 1.   , 0.27234746, 0.32236759])



Roulette Wheel Selection

```
# create proportion of selection by dividing each fit by sum
distribution = scaled_fit / np.sum(scaled_fit)
distribution
```

```
array([0.12861987, 0.12771703, 0.46632977, 0.12700373, 0.1503296 ])
```

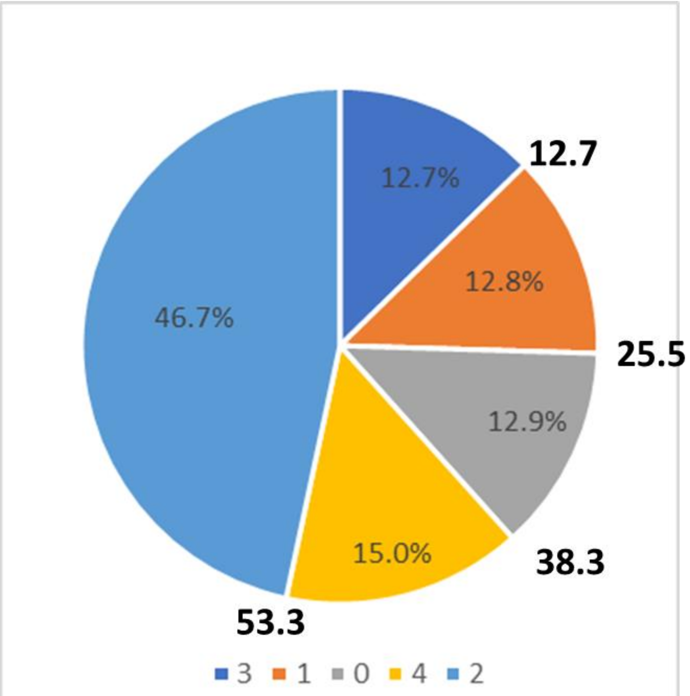



```
# instead of sorting the actual individuals, we can sort the indices
# and use those to retrieve the individuals
ind = range(0, size)  # indicies
temp = np.column_stack((distribution, ind))
temp = temp[np.argsort(temp[:,0]),]  # sort fitness values
temp  # notice that now our indices are sorted along with f
```

```
array([[0.12700373, 3.        ],
       [0.12771703, 1.        ],
       [0.12861987, 0.        ],
       [0.1503296 , 4.        ],
       [0.46632977, 2.        ]])
```

```
distribution = temp[:, 0]  # retrieve sorted fitness values
distribution
```

```
array([0.12700373, 0.12771703, 0.12861987, 0.1503296 , 0.46632977])
```




```
# create a cumulative distrubtion from our scaled proportional fitness values
cumulative_sum = np.cumsum(distribution)
cumulative_sum # this creates our roulette wheel
```

```
array([0.12700373, 0.25472076, 0.38334063, 0.53367023, 1.        ])
```

Roulette Wheel Selection

```
# create a cumulative distrubtion from our scaled proportional fitness values
cumulative_sum = np.cumsum(distribution)
cumulative_sum # this creates our roulette wheel
```

```
array([0.12700373, 0.25472076, 0.38334063, 0.53367023, 1.        ])
```

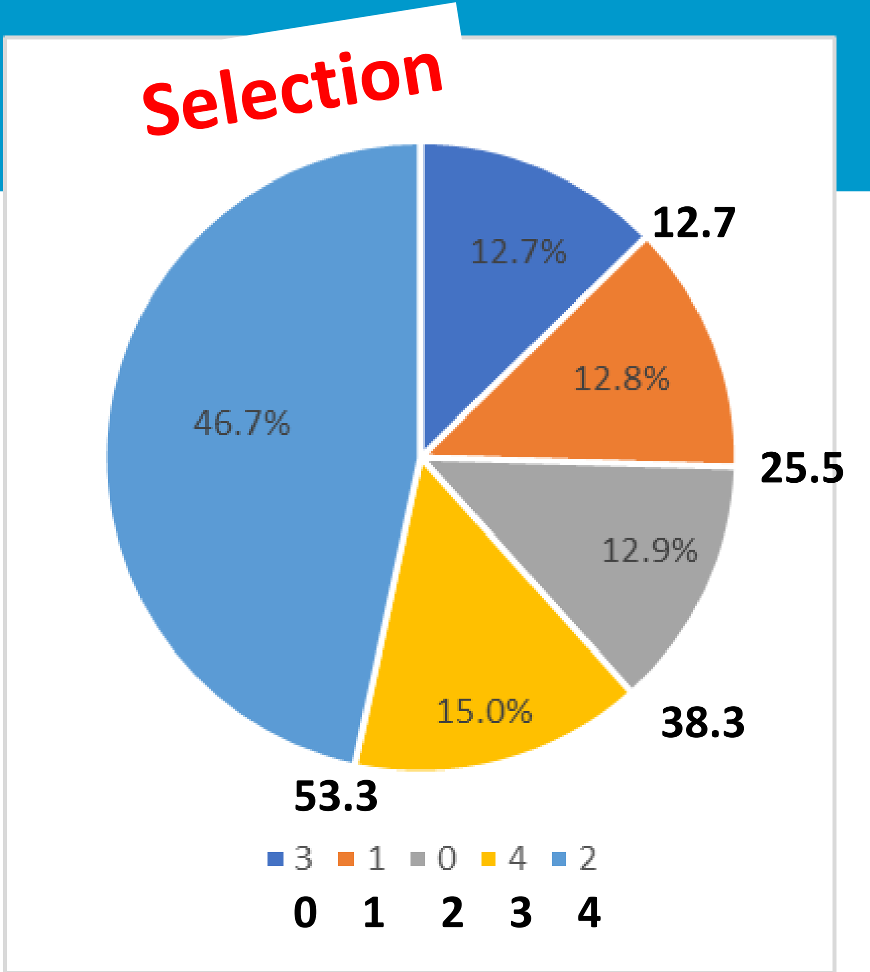
```
# Now it is time for selection, so we randomly create 5 values
# and see where they lie on the whell, AKA inbetween the
# cumulative sum above
r = np.random.uniform(0, 1, size)
r
```

```
array([0.89629309, 0.12558531, 0.20724288, 0.0514672 , 0.44080984])
```

```
sel_ind = []
for p in r: # for every random probability value in r
    index = 0
    # stop when the probablity value is greater than the value
    # in the cumulative sum
    # ex: say our cumulative sum is [ 0, 0.25, 0.6, 0.9, 1]
    # our random value is 0.43, which lies between 0.25 and 0.6;
    # therefore, index 2 is chosen
    while p > cumulative_sum[index]:
        index += 1

    sel_ind.append(index)
sel_ind  # the indicies of the selected individuals
```

```
[4, 0, 1, 0, 3]
```

Repeat to generate the 2 set of parents

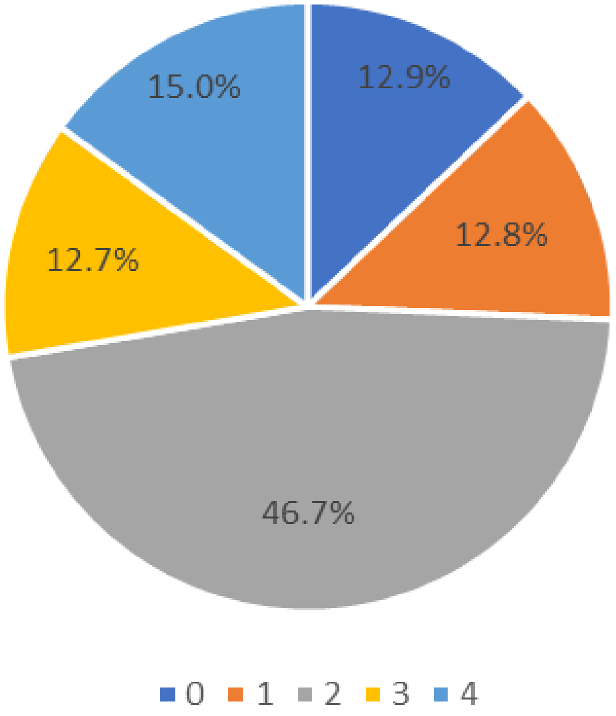


Roulette Wheel Selection

Selection

```
# now that we have the indices of our parents selected, we need
# to get their actual values
# remember 'temp' holds the original indices of our population
ind1 = np.asarray(temp[sel_ind, 1], dtype=int).tolist()
selected = next_gen[ind1]  # get selected individuals
f1 = scaled_fit[ind1]
# do the same for the second set of parents
ind2 = np.asarray(temp[mat_ind, 1], dtype=int).tolist()
mates = next_gen[ind2]
f2 = scaled_fit[ind2]
print("Parent 1", ind1)
print("X Value 1", selected) # selected individuals
print("Fitness1",f1)  # their fitness values
print("Parent 1", ind2)
print("X Value 2",mates)  # second set of parents
print("Fitness1",f2)  # their fitness values
```

```
Parent 1 [2, 3, 1, 3, 4]
X Value 1 [-8.36381044  0.43310421  8.3259129   0.43310421 15.71787817]
Fitness1 [1.         0.27234746 0.27387707 0.27234746 0.32236759]
Parent 1 [3, 4, 2, 0, 2]
X Value 2 [ 0.43310421 15.71787817 -8.36381044  2.0319161  -8.36381044]
Fitness1 [0.27234746 0.32236759 1.         0.27581313 1.        ]
```



Crossover

Crossover

```
# now it is time to create the offspring
# we will create 5 offspring to replace the five sets of parents
p_cross = 0.75  # we crossover probabilistcally
offspring = []
r = np.random.uniform(0, 1, 5)  # random values to determine to crossover
for i in range(0, size):

    if r[i] < p_cross:
        # crossover technique is average, where gamma is some random
        # value between 0 and 1
        gamma = np.random.uniform(0, 1, 1)[0]
        child = (1-gamma)*selected[i]+gamma*mates[i]
    else:
        # if we don't crossover, choose fittest parent as offspring
        if f1[i] < f2[i]:
            child = mates[i]
        else:
            child = selected[i]
    offspring.append(child)
offspring
```

$\alpha P1_i + (1 - \alpha) P2_i$

```
[-5.86966449761192,
 11.02756061658421,
 0.9748622091105337,
 0.6839062141340411,
 2.601810115896935]
```

X Values

Mutation

Mutation

```python
# now that we've crossed over the parents, we need to mutate to introduce
# new genetic material
p_mut = 0.75
# how much should we mutate by? It usually be around 1% of the
# entire domain space
bound_mut = 0.01 * (np.abs(lower_bound)+np.abs(upper_bound))
print("BOUND: {}".format(bound_mut))

r = np.random.uniform(0, 1, 5)
for i in range(0, size):
    # we either mutate or don't
    if r[i] < p_mut:
        offspring[i] += np.random.uniform(-bound_mut, bound_mut, 1)[0]

offspring = np.asarray(offspring)
offspring
```

Random value between -0.4 and 0.4

```
BOUND: 0.4

array([-5.8696645 , 11.40835695,  1.11276915,  1.0061735 ,  2.60181012])
```

X Values

Replacement - Elitism

Replacement

```
# now that we've created our new offspring, lets check out how well it
# does!
new_fit = fitness_function(offspring)
scaled_new_fit=scale_fitness_1(new_fit)
scaled_new_fit=scale_fitness_2(scaled_new_fit)
ind = range(0, size)
# combine the new fitness with the new indices
temp = np.column_stack((scaled_new_fit, ind))
temp = temp[np.argsort(-temp[:,0]),]  # sort from largest to malles
temp
```

```
array([[1.        , 1.        ],
       [0.74421311, 0.        ],
       [0.51115549, 2.        ],
       [0.4482003 , 3.        ],
       [0.35784649, 4.        ]])
```

Scaled Fitness

```
# now it is time for elitism, we need to replace the worst 20% with the
# previous generation's best 20%
ind_replace = np.array(temp[best_index:size, 1], dtype=int).tolist()
ind_replace  # this is the set of indices from the preivous gen t
```

```
[4]
```

Index of worst solution

Replacement - Elitism

Replacement

```
array([-5.8696645 , 11.40835695,  1.11276915,  1.0061735 ,  2.60181012])
```

```
# now that we've created our new offspring, lets check out how w  l it
# does!
new_fit = fitness_function(offspring)
scaled_new_fit=scale_fitness_1(new_fit)
scaled_new_fit=scale_fitness_2(scaled_new_fit)
ind = range(0, size)
# combine the new fitness with the new indices
temp = np.column_stack((scaled_new_fit, ind))
temp = temp[np.argsort(-temp[:,0]),]  # sort from largest to ma   s
temp
```

```
array([[1.        , 1.        ],
       [0.74421311, 0.        ],
       [0.51115549, 2.        ],
       [0.4482003 , 3.        ],
       [0.35784649, 4.        ]])
```

```
# now it is time for elitism, we need to replace the worst 20% w  h the
# previous generation's best 20%
ind_replace = np.array(temp[best_index:size, 1], dtype=int).tol  ()
ind_replace  # this is the set of indices from the preivous gen
```

```
[4]
```

```
next_gen = np.copy(offspring)
next_gen[ind_replace] = prev_gen[np.asarray(best_ind, dtype=int) olist()]
next_gen  # here we have our new generation
```

```
array([-5.8696645 , 11.40835695,  1.11276915,  1.0061735 , -8.36381044])
```

Generational replacement with Elitism

```
fitness=fitness_function(next_gen)
fitness
```

```
array([-0.25305323, -0.59675436,  0.35959753,  0.63439095, -1.26017642])
```

Repeat

```python
iteration=0
while iteration<10:
    prev_gen=np.copy(next_gen)
    fitness = fitness_function(next_gen)
    # because we are using roulette wheel selection, we cannot sum up negative
    # fitness values; therefore, we need to scale so that all fitness values are
    # positive; this is done by adding the absolute value of the minimum to each
    # value; here is our new scaled fitness
    scaled_fit=scale_fitness_1(fitness) # scale to make all positive
    #If we are maximizing, we don't perform this second layer of scaling,
    #but if we are then we scale using the equation below.
    scaled_fit=scale_fitness_2(scaled_fit)  # scale for minimization
    # create proportion of selection by dividing each fit by sum
    distribution = scaled_fit / np.sum(scaled_fit)
    # instead of sorting the actual individuals, we can sort the indices
    # and use those to retrieve the individuals
    ind = range(0, size)  # indicies
    temp = np.column_stack((distribution, ind))
    temp = temp[np.argsort(temp[:,0]),]  # sort fitness values
    distribution = temp[:, 0]  # retrieve sorted fitness values
    # now that our fitness values are sorted, we can take the best 20%
    elitism = 0.2
    best_index = size-int(size*elitism)
    # index in our indices where the best 20% start
    best_ind = range(best_index, size)
    best_ind = temp[best_ind, 1]
    # create a cumulative distrubtion from our scaled proportional fitness values
    cumulative_sum = np.cumsum(distribution)
    # Now it is time for selection, so we randomly create 5 values
    # and see where they lie on the whell, AKA inbetween the
    # cumulative sum above
    r = np.random.uniform(0, 1, size)
    sel_ind = []
    for p in r: # for every random probability value in r
        index = 0
        # stop when the probablity value is greater than the value
        # in the cumulative sum
        # ex: say our cumulative sum is [ 0, 0.25, 0.6, 0.9, 1]
        # our random value is 0.43, which lies between 0.25 and 0.6;
        # therefore, index 2 is chosen
        while p > cumulative_sum[index]:
```

...

```
Generation 1
  Mean : -0.22319910762638862
  Best: -1.2601764246198135
Generation 2
  Mean : -0.5842603558692865
  Best: -1.3778888937046347
Generation 3
  Mean : -1.0497412189377808
  Best: -1.9358380156857249
Generation 4
  Mean : -1.45655822082373
  Best: -1.9358380156857249
Generation 5
  Mean : -1.5360473776407404
  Best: -1.986779447323701
Generation 6
  Mean : -1.8145848415475396
  Best: -1.986779447323701
Generation 7
  Mean : -1.6897619483576611
  Best: -1.9872100598662024
Generation 8
  Mean : -1.9367517983204805
  Best: -1.9872100598662024
Generation 9
  Mean : -1.9706295578846365
  Best: -1.9873243587182432
Generation 10
  Mean : -1.9723588808127652
  Best: -1.9874021181112054
```

Evolutionary Algorithm in Python

Nuno Antunes Ribeiro

Assistant Professor

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Engineering Systems and Design

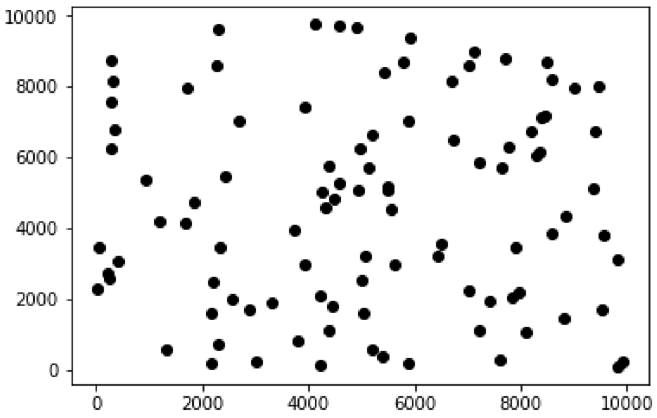
TSP Instance

Generate and Process Instance Data

```
#Generate Data Inputs

# Select random seed
random.seed(1)

# Number of cities
n=100

#Coordinate Range
rangelct=10000

#No. of swaps at each iteration
no_swap=1

#Generate random locations
coordlct_x = random.choices(range(0, rangelct), k=n)
coordlct_y = random.choices(range(0, rangelct), k=n)
```

Same seed – same instances solved using local search metaheuristics

```
plt.plot(coordlct_x, coordlct_y, 'o', color='black');
```



Object-Oriented Programming

- In object-oriented programming we can bind data and functions together in a same class of objects
- A city in the TSP is an object with data concerning the corresponding coordinates and functions (methods) to compute distance between city objects

Understanding Object Oriented Programming:
https://www.youtube.com/watch?v=wfcWRAxRVBA

```
class City:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, city):
        return math.hypot(self.x - city.x, self.y - city.y)

    def __repr__(self):
        return f"({self.x}, {self.y})"

cities = []
for line in range(n):
    cities.append(City(coordlct_x[line], coordlct_y[line]))
```

```
cities
```

```
[(1343, 561),
 (8474, 8700),
 (7637, 5699),
 (2550, 1998),
 (4954, 5047),
 (4494, 4849),
 (6515, 3567),
 (7887, 3460),
 (938, 5384),
 (283, 6234),
 (8357, 6124),
 (4327, 4581),
```

```
#Compute Distance Between Cities
City.distance(cities[1],cities[2])
```

```
3115.5368718729683
```

Generate Initial Population

Initial Population

- **2-approaches to generate the initial population:**
  - Completely at random (good to ensure diversity)
  - Greedy approach (initiate the search with good solutions)

```
#Function to generate a completely random route
def random_route():
    return random.sample(cities, len(cities))
```

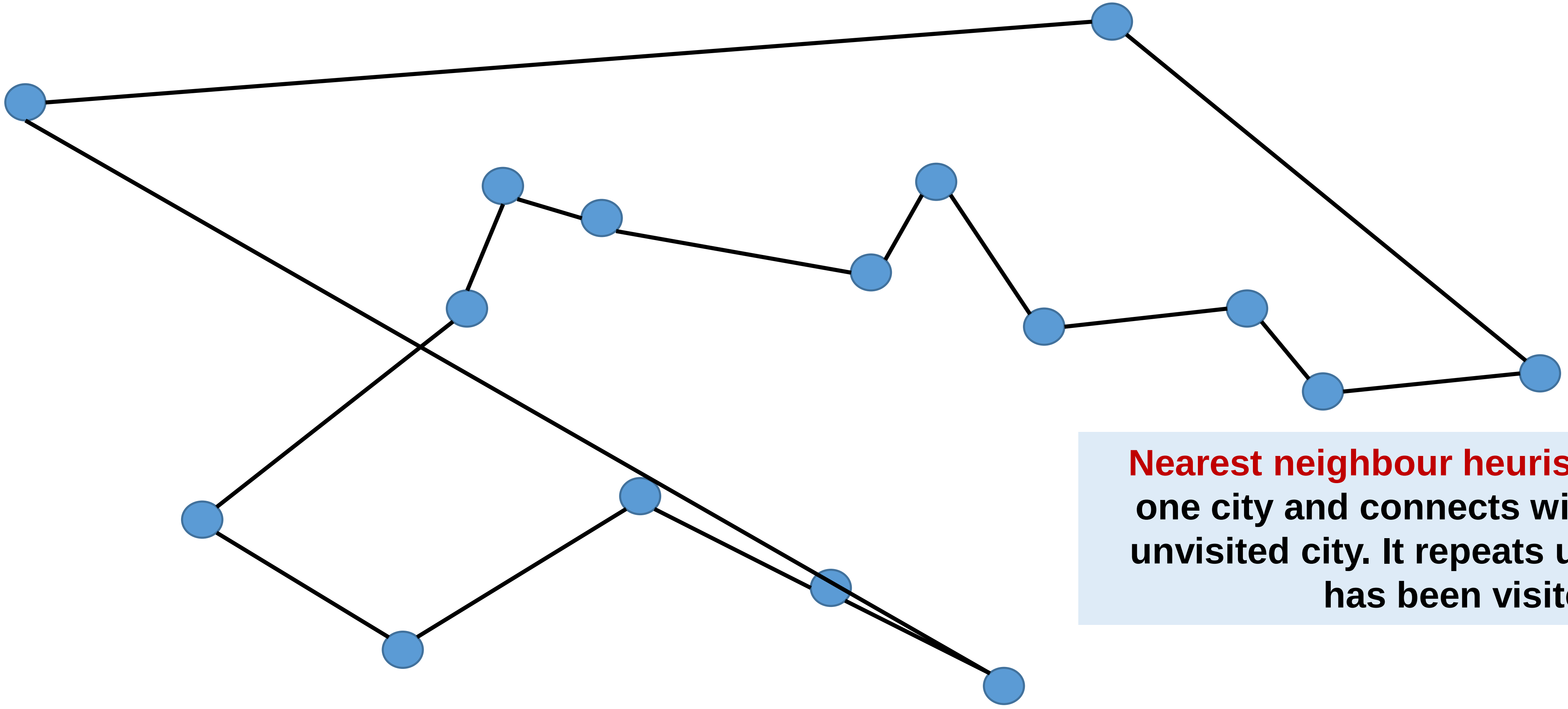
Function to generate random route

```
#Funtion to generate route using greedy approach
def greedy_route(start_index, cities):
    unvisited = cities[:]
    del unvisited[start_index]
    route = [cities[start_index]]
    while len(unvisited):
        index, nearest_city = min(enumerate(unvisited), key=lambda num: num[1].distance(route[-1]))
        route.append(nearest_city)
        del unvisited[index]
    return route
```

Function to generate greedy route

Greedy Approach

Initial Population



Nearest neighbour heuristic: It starts at one city and connects with the closest unvisited city. It repeats until every city has been visited.

Greedy Approach

Initial Population

```
#Funtion to generate route using greedy approach
def greedy_route(start_index, cities):
    unvisited = cities[:]
    del unvisited[start_index]
    route = [cities[start_index]]
    while len(unvisited):
        index, nearest_city = min(enumerate(unvisited), key=lambda num: num[1].distance(route[-1]))
        route.append(nearest_city)
        del unvisited[index]
    return route
```

Delete start city from unvisited list

While there are still cities unvisited, compute the nearest city from the last city visited

Delete this city from unvisited list

```
#generate greedy route starting in city with index city_index
city_index=5
greedroute=greedy_route(city_index,cities)
```

Greedy route starting from city 5

```
greedroute
```

```
[(4494, 4849),
 (4260, 4997),
 (4327, 4581),
 (4596, 5273),
 (4954, 5047),
 (5479, 5088),
 (5487, 5165),
 (5564, 4547),
 (5137, 5702),
```

Greedy Approach

Initial Population

```
#Function to compute the total distance of a route
def path_cost(route):
    return sum([city.distance(route[index - 1]) for index, city in enumerate(route)])
```

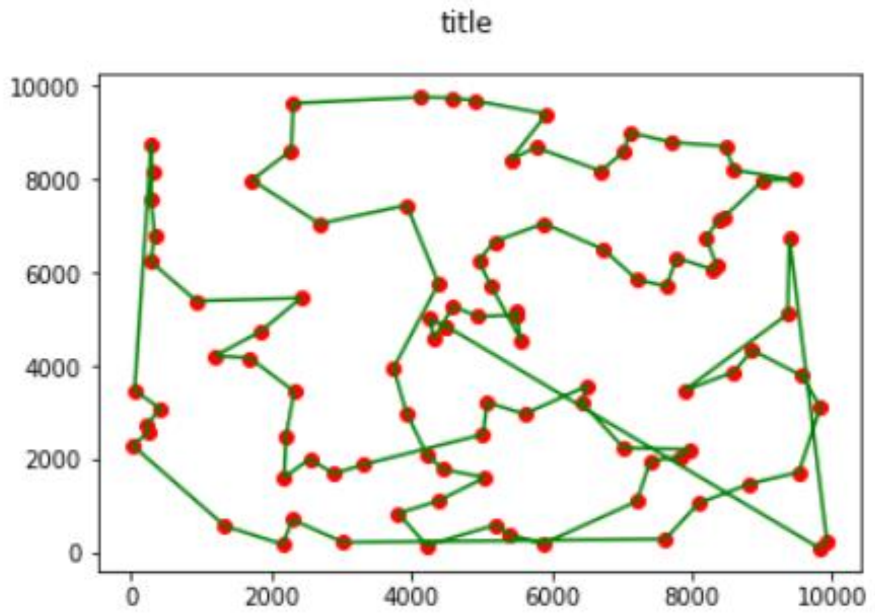
Compute total distance of the greedy route

```
#Let's compute total distance of the greedy route generated
path_cost(greedroute)
```

```
99406.3594311837
```

```
#Function to plot the routes
def visualize_tsp(title, cities):
    fig = plt.figure()
    fig.suptitle(title)
    x_list, y_list = [], []
    for city in cities:
        x_list.append(city.x)
        y_list.append(city.y)
    x_list.append(cities[0].x)
    y_list.append(cities[0].y)

    plt.plot(x_list, y_list, 'ro')
    plt.plot(x_list, y_list, 'g')
    plt.show(block=True)
```

```
visualize_tsp('title', greedroute)
```



Generate Initial Population

Initial Population

- **2-approaches to generate the initial population:**
  - Completely at random (good to ensure diversity)
  - Greedy approach (initiate the search with good solutions)

```
#Function to generate initial population
def initial_population():
    p1 = [random_route() for _ in range(population_size - greedy_seed)] # Generate n-g random routes (where n is the p
    greedy_population = [greedy_route(start_index % len(cities), cities) # Generate g routes through greedy procedure
                         for start_index in range(greedy_seed)]
    return [*p1, *greedy_population]
```

Generate $n - g$ random routes and g greedy routes

```
# Generate n-g random routes (where n is the population and g the number of routes generated using greedy approach)
p1 = [random_route() for _ in range(population_size - greedy_seed)]
p1
```

Compute Fitness of a Route

Compute Fitness

```
#Funtion to compute the fitness value
class Fitness:
    def __init__(self, route):
        self.route = route
        self.distance = 0
        self.fitness = 0.0

    def path_cost(self):
        if self.distance == 0:
            distance = 0
            for index, city in enumerate(self.route):
                distance += city.distance(self.route[(index + 1) % len(self.route)])
            self.distance = distance
        return self.distance

    def path_fitness(self):
        if self.fitness == 0:
            self.fitness = 1 / float(self.path_cost())
        return self.fitness
```

Fitness is an object with three variables: route, distance and fitness

Function (method) used to compute the **distance of the route**

Scale the distance

Recall: If our objective is to minimize a given fitness function, then we need to scale also for minimization

```
#generate object fitness for route greedroute
fitroute=Fitness(greedroute)
```

```
#Compute distance --- same value as using the path_cost function outside Fitness class
fitroute.path_cost()
```

```
99406.35943118374
```

```
# scale fitness function
fitroute.path_fitness()
```

```
1.005971857054349e-05
```

Parent Selection

Selection

- 2-approaches to select parents for reproduction (only 1 is selected):
  - **Roulette Selection**
  - Completely at random

```
def selection(self):
    selections = [self.ranked_population[i][0] for i in range(self.elites_num)]
    if self.roulette_selection:
        df = pd.DataFrame(np.array(self.ranked_population), columns=["index", "fitness"])
        df['cum_sum'] = df.fitness.cumsum()
        df['cum_perc'] = 100 * df.cum_sum / df.fitness.sum()
        for _ in range(0, self.population_size - self.elites_num):
            pick = 100 * random.random()
            for i in range(0, len(self.ranked_population)):
                if pick <= df.iat[i, 3]:
                    selections.append(self.ranked_population[i][0])
                    break
    else:
        for _ in range(0, self.population_size - self.elites_num):
            pick = random.randint(0, self.population_size - 1)
            selections.append(self.ranked_population[pick][0])
    self.population = selections
```

Compute **cumulative percentages** (slice of the roulette)

Parent Selection

Selection

- 2-approaches to select parents for reproduction (only 1 is selected):
  - **Roulette Selection**
  - Completely at random

	index	fitness	cum_sum	cum_perc
0	[(1343, 561), (2165, 166), (2308, 695), (3033,...	1.05189e-05	1.05189e-05	5.20486
1	[(3932, 2986), (7887, 3460), (2216, 2495), (99...	2.20295e-06	1.27219e-05	6.2949
2	[(3932, 2986), (5487, 5165), (938, 5384), (847...	2.18597e-06	1.49078e-05	7.37654
3	[(9831, 3103), (7887, 3460), (9452, 7984), (65...	2.15209e-06	1.70599e-05	8.44142
4	[(4221, 145), (4591, 9737), (5052, 1602), (185...	2.12764e-06	1.91876e-05	9.4942
5	[(6515, 3567), (5396, 379), (4327, 4581), (438...	2.12384e-06	2.13114e-05	10.5451
6	[(8599, 3865), (5022, 2523), (5875, 178), (837...	2.10426e-06	2.34156e-05	11.5863
7	[(305, 8164), (2187, 1596), (5052, 1602), (221...	2.10286e-06	2.55185e-05	12.6268
8	[(57, 3469), (295, 8755), (7974, 2205), (8474,...	2.08291e-06	2.76014e-05	13.6575
9	[(7974, 2205), (2550, 1998), (7033, 8585), (93...	2.0791e-06	2.96805e-05	14.6862
10	[(57, 3469), (4221, 145), (8357, 6124), (7705,...	2.06536e-06	3.17459e-05	15.7082
11	[(5487, 5165), (5479, 5088), (2308, 695), (495...	2.05861e-06	3.38045e-05	16.7268
12	[(57, 3469), (8357, 6124), (1208, 4209), (7215...	2.05437e-06	3.58589e-05	17.7433
13	[(4596, 5273), (5022, 2523), (2897, 1681), (83...	2.03141e-06	3.78903e-05	18.7485
14	[(3935, 7438), (8861, 4329), (7405, 1941), (50...	2.02895e-06	3.99192e-05	19.7524
15	[(8474, 8700), (295, 8755), (3935, 7438), (882...	2.01369e-06	4.19329e-05	20.7488

...

Compute **cumulative percentages** (slice of the roulette)

Parent Selection

Selection

- 2-approaches to select parents for reproduction (only 1 is selected):
  - **Roulette Selection**
  - Completely at random

```
def selection(self):
    selections = [self.ranked_population[i][0] for i in range(self.elites_num)]
    if self.roulette_selection:
        df = pd.DataFrame(np.array(self.ranked_population), columns=["index", "fitness"])
        df['cum_sum'] = df.fitness.cumsum()
        df['cum_perc'] = 100 * df.cum_sum / df.fitness.sum()
        for _ in range(0, self.population_size - self.elites_num):
            pick = 100 * random.random()
            for i in range(0, len(self.ranked_population)):
                if pick <= df.iat[i, 3]:
                    selections.append(self.ranked_population[i][0])
                    break
    else:
        for _ in range(0, self.population_size - self.elites_num):
            pick = random.randint(0, self.population_size - 1)
            selections.append(self.ranked_population[pick][0])
    self.population = selections
```

Randomly pick a number between 0 and 100;
Select the parent that is ranked in the generated picked position

Crossover Operator

Crossover

```python
#Crossover Operator
def produce_child(parent1, parent2):
    gene_1 = random.randint(0, len(parent1))
    gene_2 = random.randint(0, len(parent1))
    gene_1, gene_2 = min(gene_1, gene_2), max(gene_1, gene_2)
    child = [parent1[i] for i in range(gene_1, gene_2)]
    child.extend([gene for gene in parent2 if gene not in child])
    return child
```

Parents

Gene 1 Gene 2

1	4	3	7	5	9	6	2	8	10	Parent 1
3	7	10	8	5	4	6	2	1	9	Parent 2
3	7	5	9	10	8	4	6	2	1	Child

Crossover Operator

Crossover

```
#Generate Children through crossover
#We assume that the top n routes (elites_num) are mantained from iteration to iteration (those belong to the elite)
#Therefore we just need to generate the p routes , i.e. p=(length of population - elites_num)
def generate_population(self):
    length = len(self.population) - self.elites_num
    children = self.population[:self.elites_num]
    for i in range(0, length):
        child = self.produce_child(self.population[i],
                                   self.population[(i + random.randint(1, self.elites_num)) % length])
        children.append(child)
    return children
```

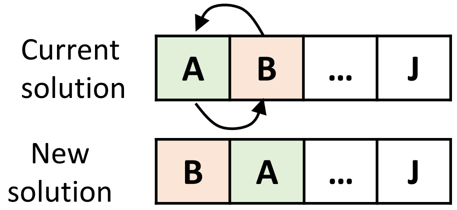
We generate n=(Population – No. Elite Individuals) solutions

Mutation Operator

Mutation

- Randomly select 2 cities to swap

Swap operator




```
def mutate(self, individual):
    if self.swap_operator==1:
    #Swap Operator
        for index, city in enumerate(individual):
            if random.random() < max(0, self.mutation_rate):
                random_index = random.sample(range(len(individual)), 1)
                individual[index], individual[random_index[0]] = individual[random_index[0]], individual[index]
        return individual
```

Next Generation

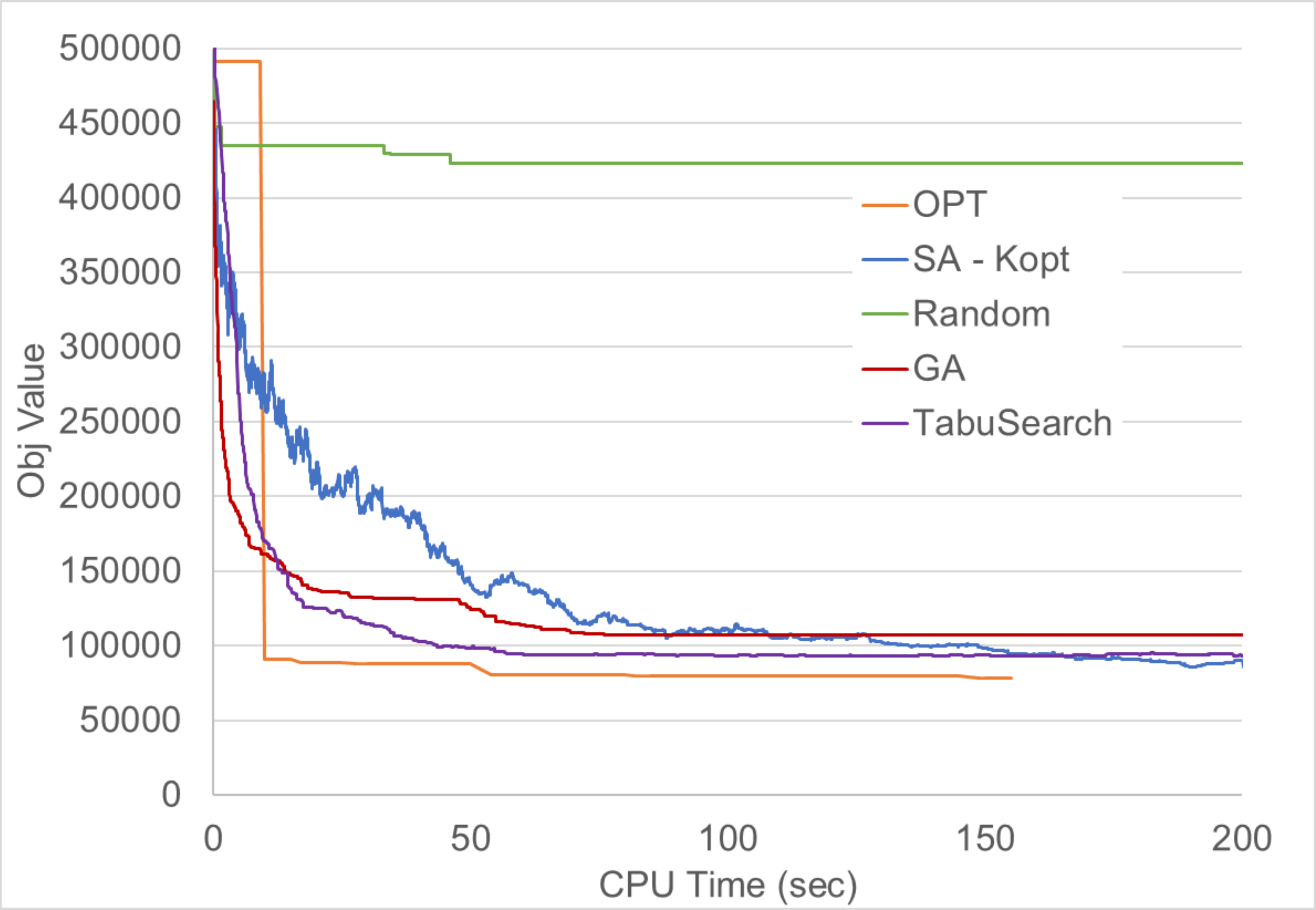
Replacement

Generational replacement with Elitism

```
def next_generation(self):
    self.rank_population()
    self.selection()
    self.population = self.generate_population()
    self.population[self.elites_num:] = [self.mutate(chromosome)
                                        for chromosome in self.population[self.elites_num:]] #We just apply mutation to the chi
```

```
def run(self):
    if self.plot_progress:
        plt.ion()
    for ind in range(0, self.iterations):
        self.next_generation() # apply next generation function every iteration
        self.progress.append(self.best_distance()) #save the best distance found
        if self.plot_progress and ind % 10 == 0: #plot at iterations that are multiple of 10
            self.plot()
            print(ind)
        elif not self.plot_progress and ind % 10 == 0:
            print(self.best_distance())
```

TSP n=100



Obj Value
500000
450000
400000
350000
300000
250000
200000
150000
100000
50000
0
0
50
100
150
200
CPU Time (sec)
OPT
SA - Kopt
Random
GA
TabuSearch