Evolutionary Algorithms (Review)

Nuno Antunes Ribeiro

Assistant Professor

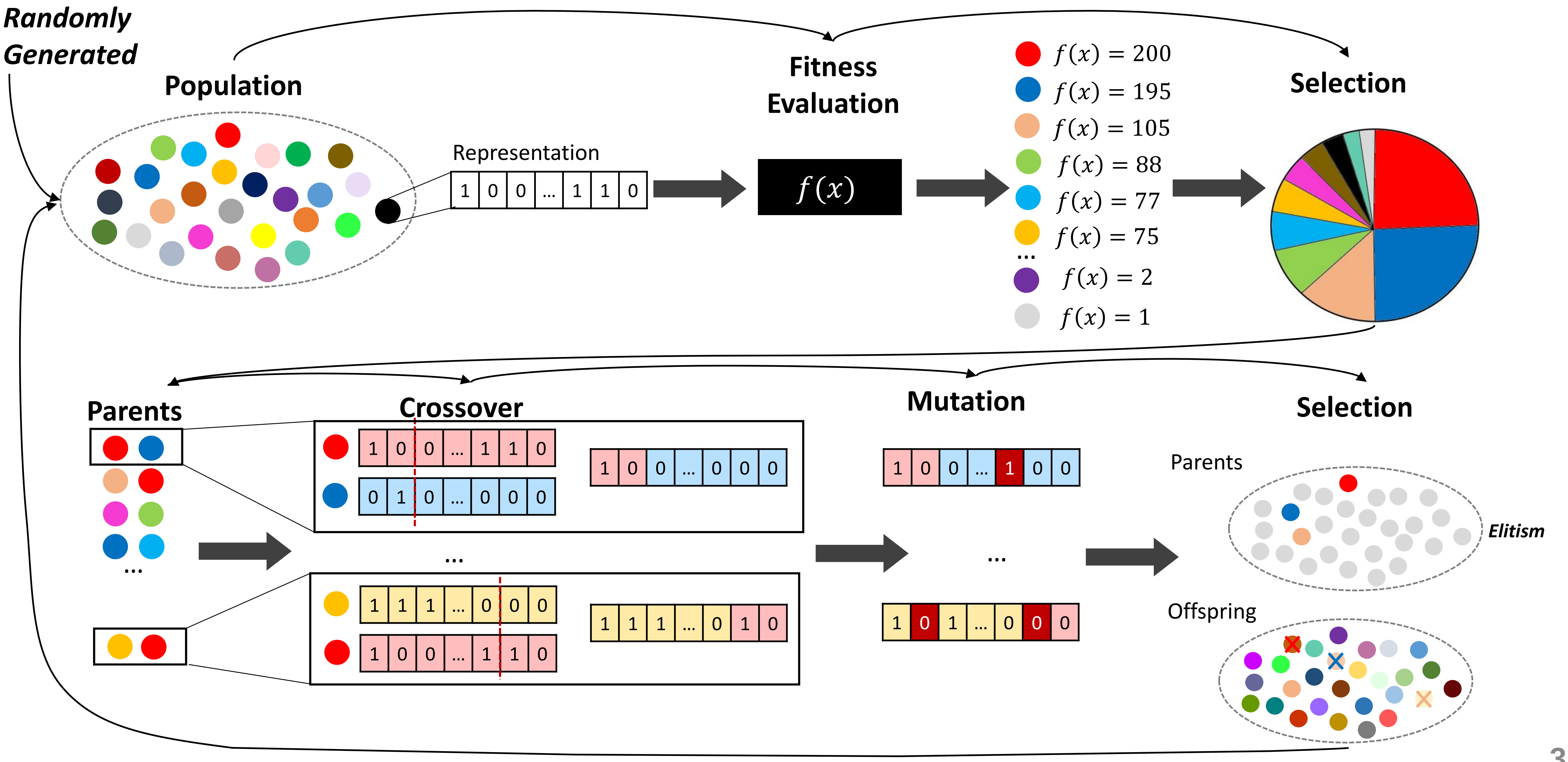
SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Engineering Systems and Design

Course Schedule

Week	Session 1	Session 2	Assignment
1	Introduction to Metaheuristic Optimization; NP-Hard models;	Exhaustive Search Methods and Backtracking; Branch and Bound; Solving Optimization Problems with Pyomo	
2	**Chinese New Year**	Random Sampling ; Local Search	HA1 – LS algorithms
3	Solution Encoding; Move Operators	Escaping Local Optima ; Simulated Annealing	
4	Variable Neighborhood Search; Greedy Constructive Heuristics	Tabu Search	
5	Common Concepts for Metaheuristics; Comparing Optimization Algorithms	Very Large Neighborhood Search (VLNS);	
6	Introduction to Evolutionary Algorithms	Project Consultation	HA2 – EA algorithms
7	**BREAK**		
8	Types of Evolutionary Algorithms	Using a Genetic Algorithm to Calibrate Neural Networks No Free Lunch Theorem	
9	Genetic Programming	Neuroevolution Multi-Objective Optimization;	
10	NSGA-II – Application Example	Particle Swarm Optimization	HA3 – Swarm algorithms
11	Ant Colony Optimization	Other Swarm Optimization Algorithms	
12	Project Consultation	Project Consultation	
13	Project Consultation	Project Presentations	
14	Final Exam		

Evolutionary Algorithm (Review)

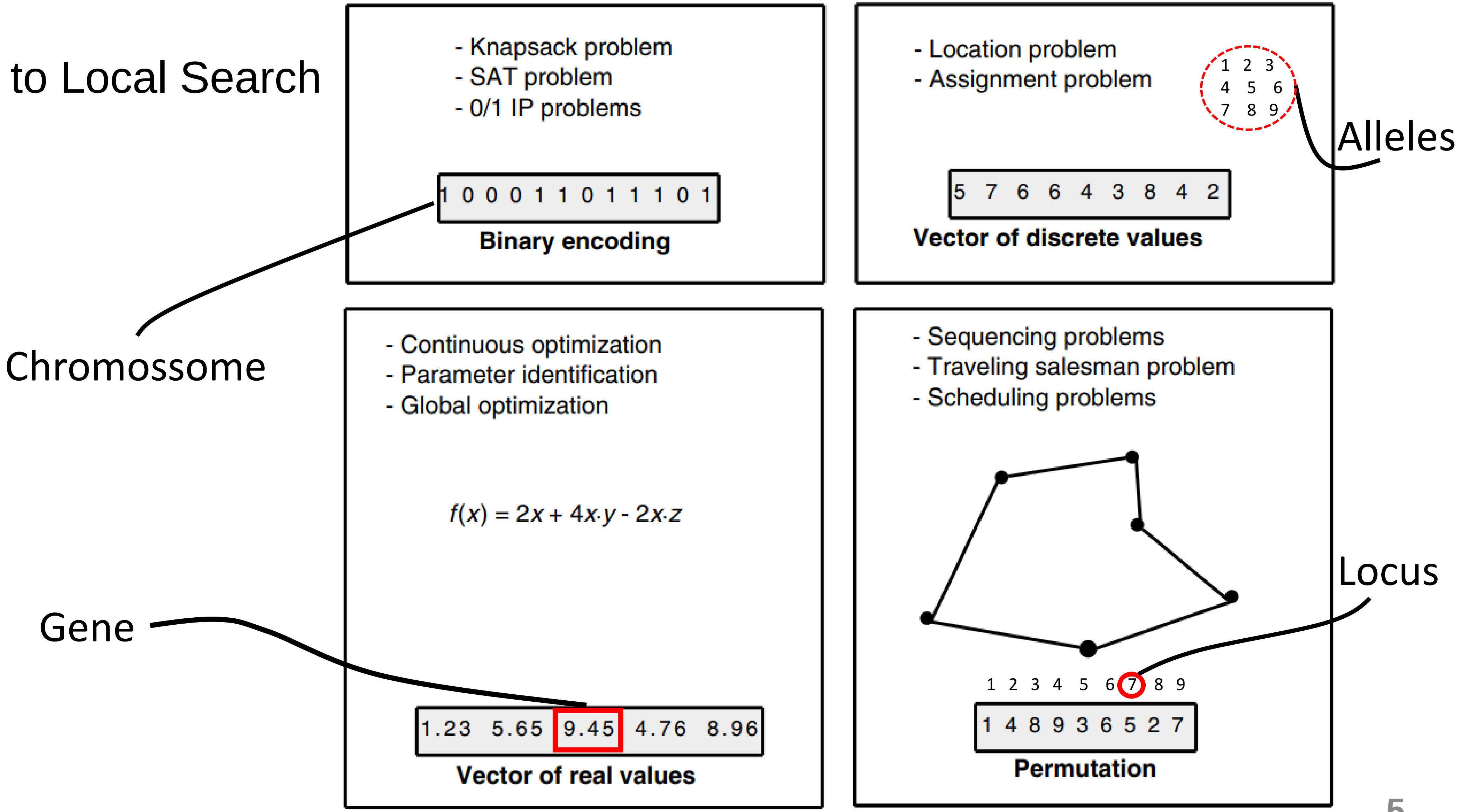


Common Concepts in EA

- **Representation:** Similarly to local Search algorithms, solutions are encoded. The encoded solutions are referred as **chromosomes**, 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

Representation

- Similar to Local Search



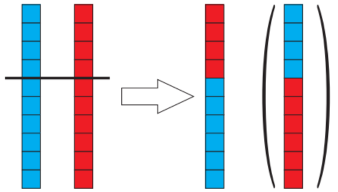
Selection Strategy

- 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.

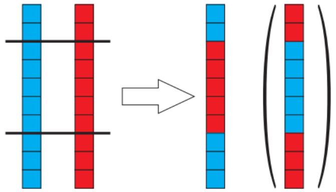
Crossover Operators

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

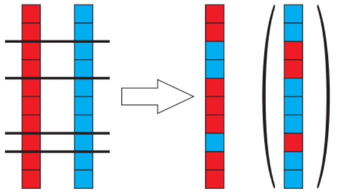
- **Discrete and Binary Operators**



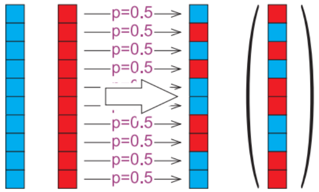
Single-Point (SPX)



Two-Point (TPX)



Multi-Point (MPX)



Uniform (UX)

- **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

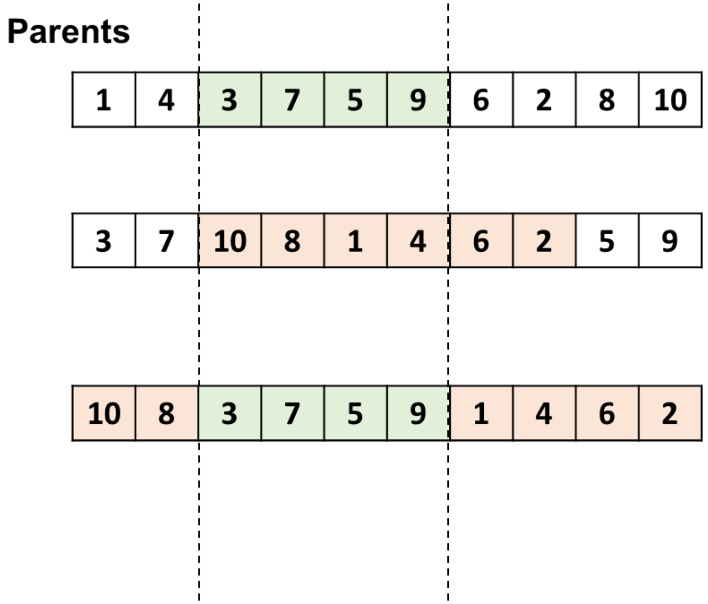
0.14	0.29	0.28	0.49	0.38	0.23	0.60	0.16	0.29	0.91	e.g. $\alpha = 0.3$

Crossover Operators

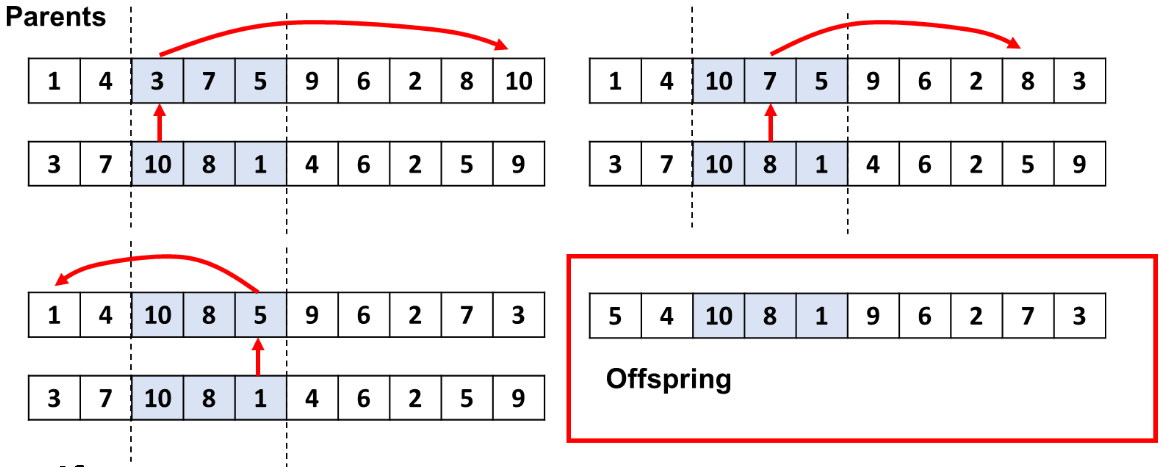
- There exist different crossover operators. As for the mutation operator, the design of crossover operators mainly depends on the representation used

- **Permutation Operators**

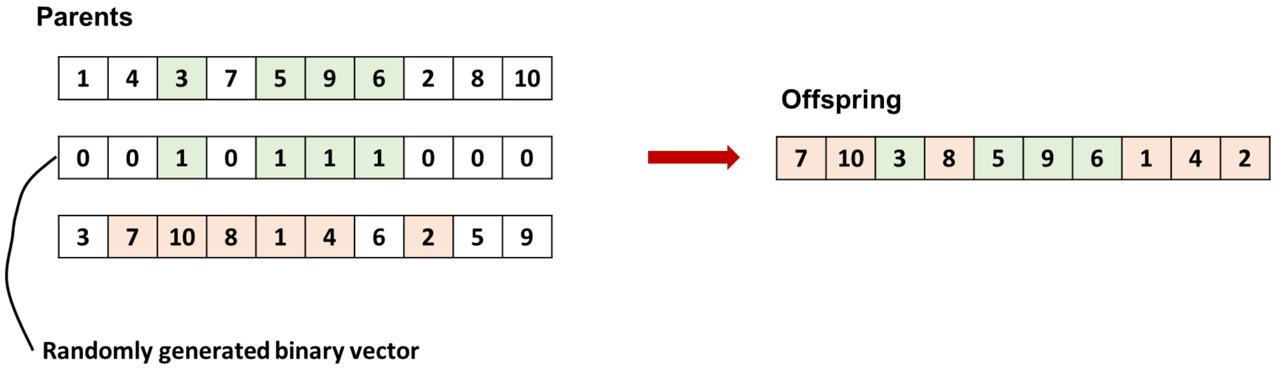
Order Crossover



Partially Mapped Crossover



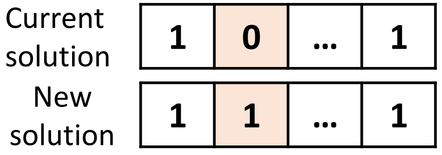
Uniform Crossover



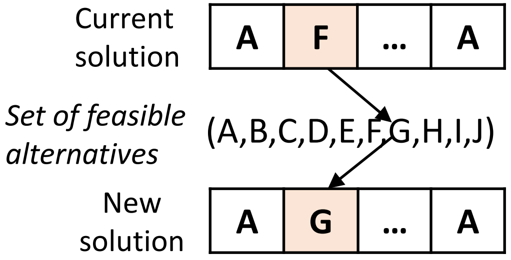
Mutation Operators

- 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

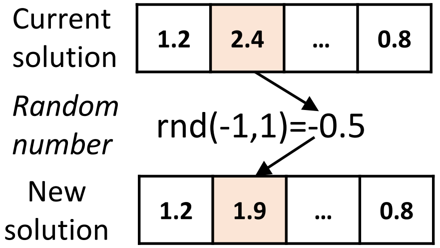
Binary encoding – flip **n** bits of the solution (typically 1 or 2 bits)



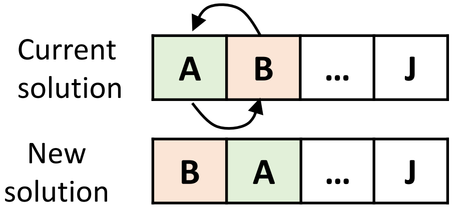
Discrete encoding – update **n** bits of the solution by randomly generating a new value (typically 1 or 2 bits)



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)



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.

- **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.

Evolutionary Algorithm in Python

Nuno Antunes Ribeiro

Assistant Professor

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Engineering Systems and Design

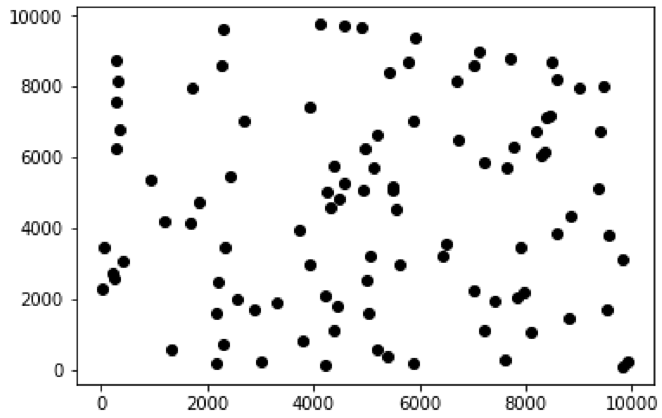
TSP Instance

Generate and Process Instance Data

```
#Generate Data Inputs

# Select random see
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

```python
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]))
```

```python
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),
```

```python
#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))
```

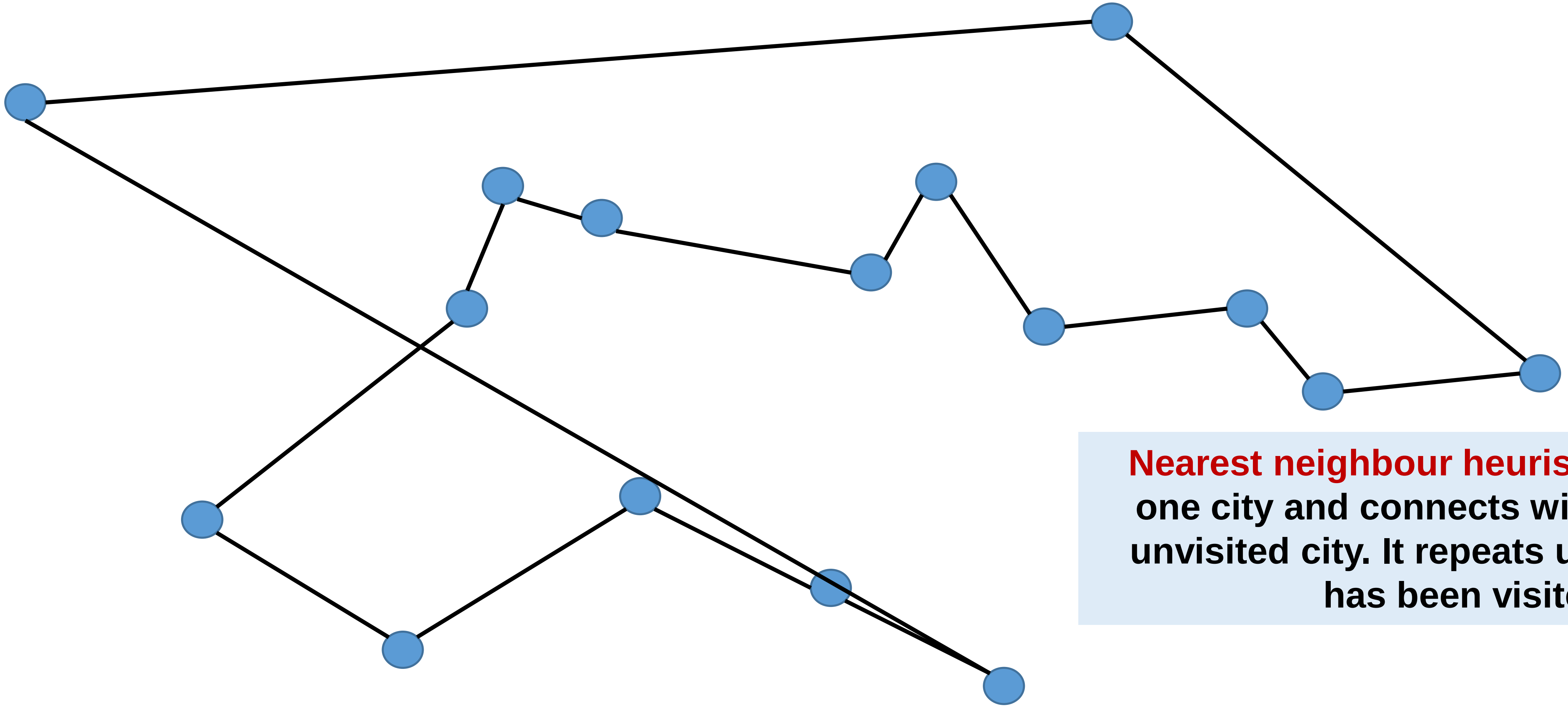
Function to generate random route

```
#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)])
```

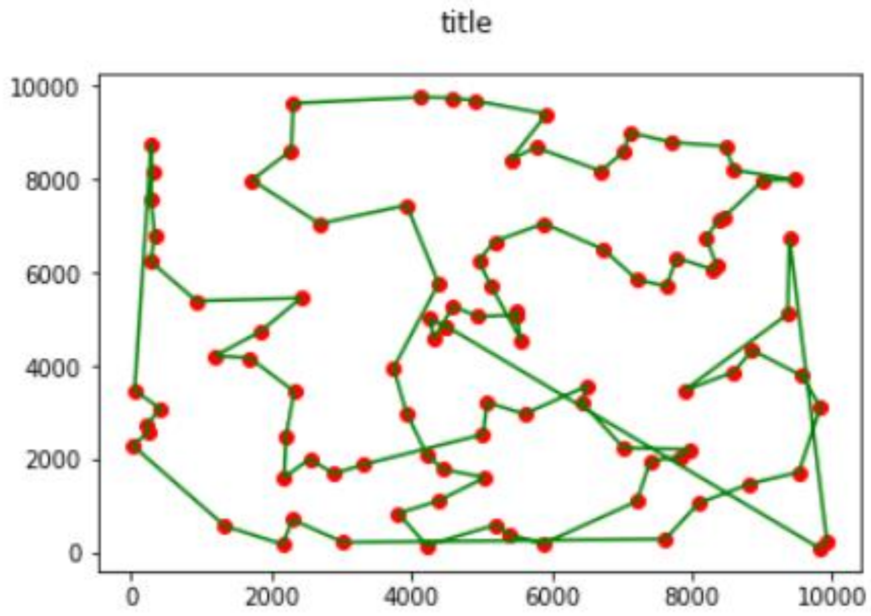
Compute total distance of the greedy route

```
#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
```

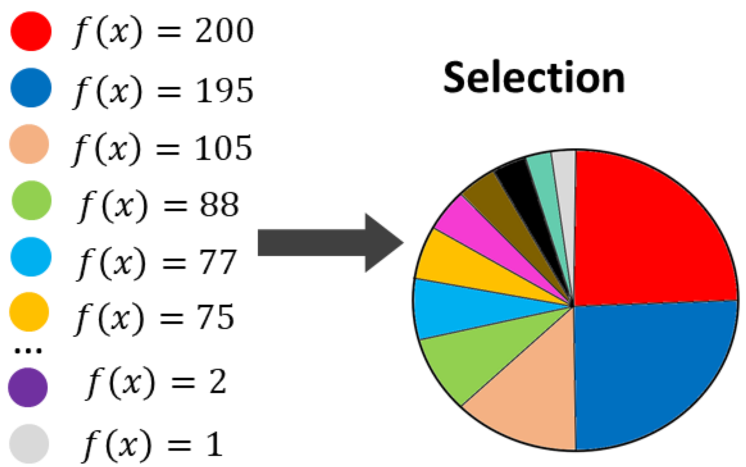
Parent Selection

Selection

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

```python
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

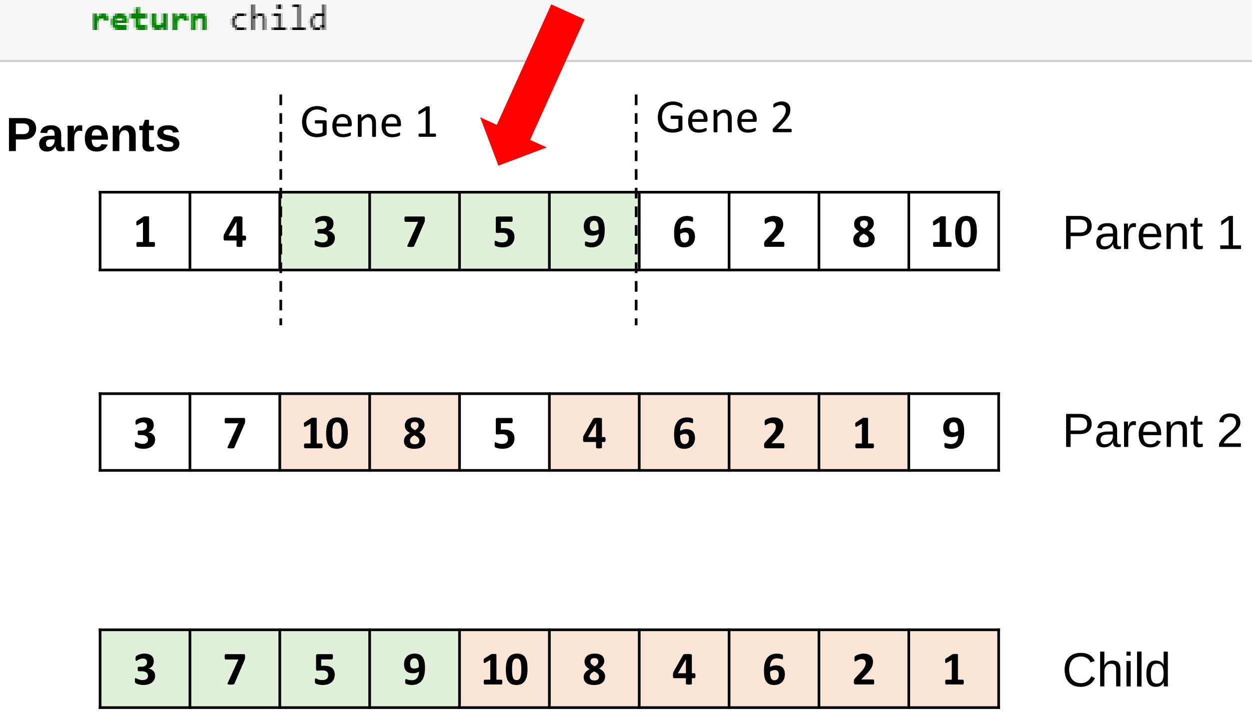
```
def selection(self):
    selections = [self.ranked_population[i][0] for i in range(self.elites_num)]
    if self.roulette_selection:
        df = pd.DataFrame(np.array(self.ranked_population), columns=["index", "fitness"])
        df['cum_sum'] = df.fitness.cumsum()
        df['cum_perc'] = 100 * 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
```



Crossover Operator

Crossover

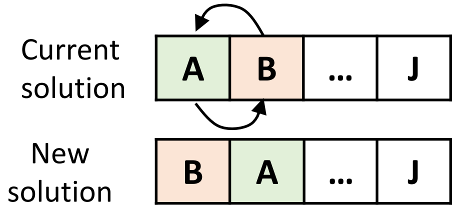
```
#Generate Children through crossover
#We assume that the top n routes (elites_num) are mantained from iteration to iteration (those belong to the elite)
#Therefore we just need to generate the p routes , i.e. p=(length of population - elites_num)
def generate_population(self):
    length = len(self.population) - 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

Swap operator



- Randomly select 2 cities to swap

```
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

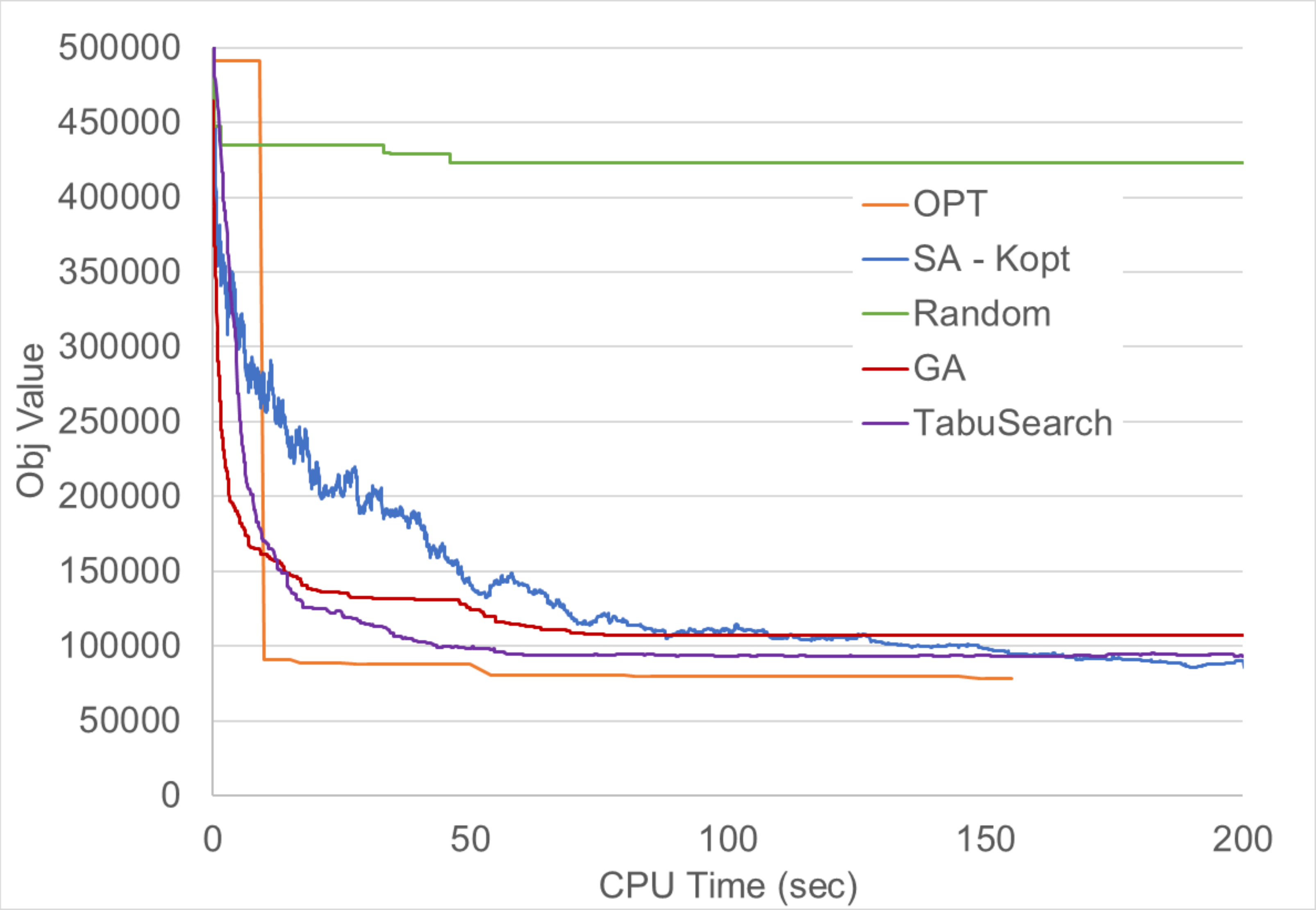
Replacement

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

Activity 2

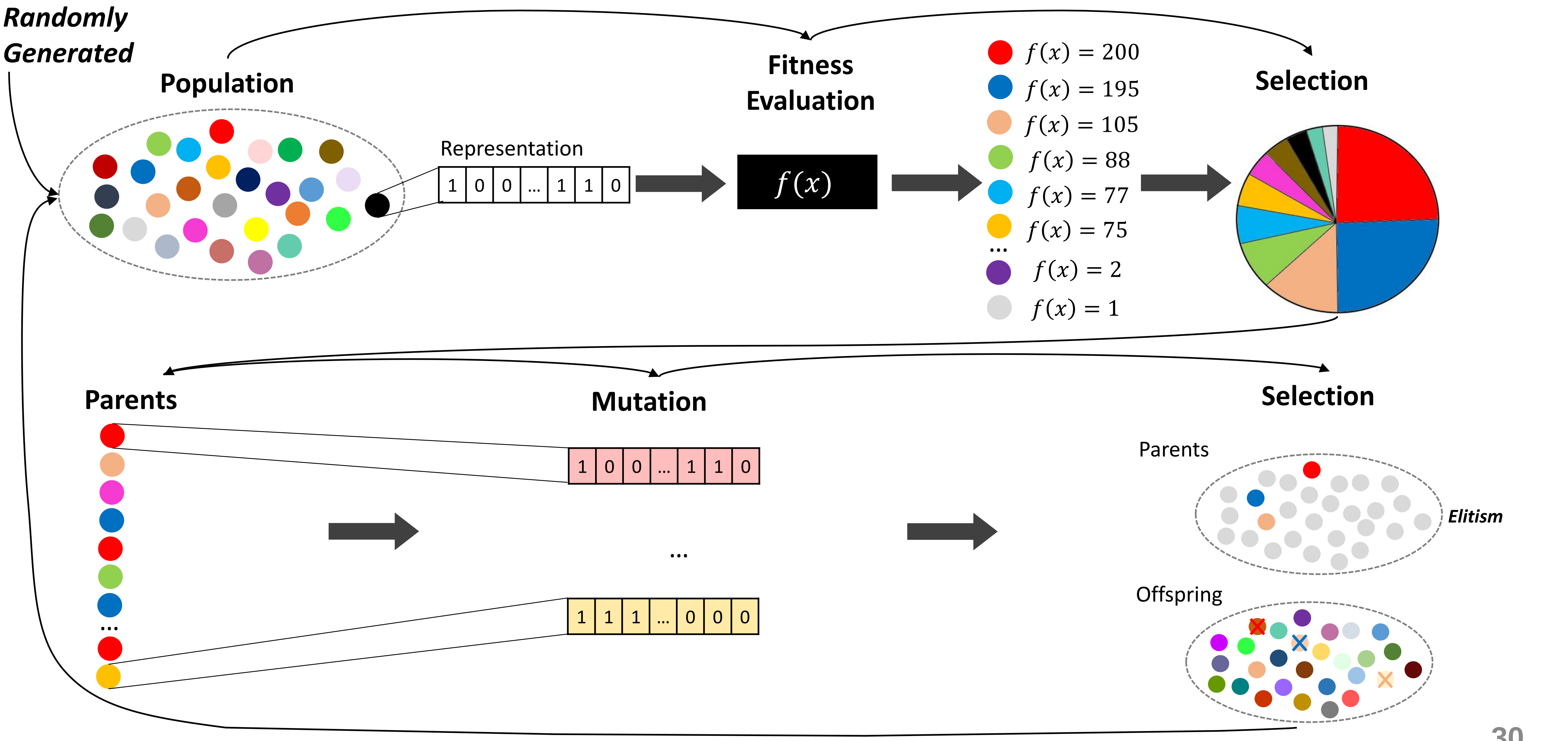
Nuno Antunes Ribeiro
Assistant Professor

Engineering Systems and Design

Activity 2

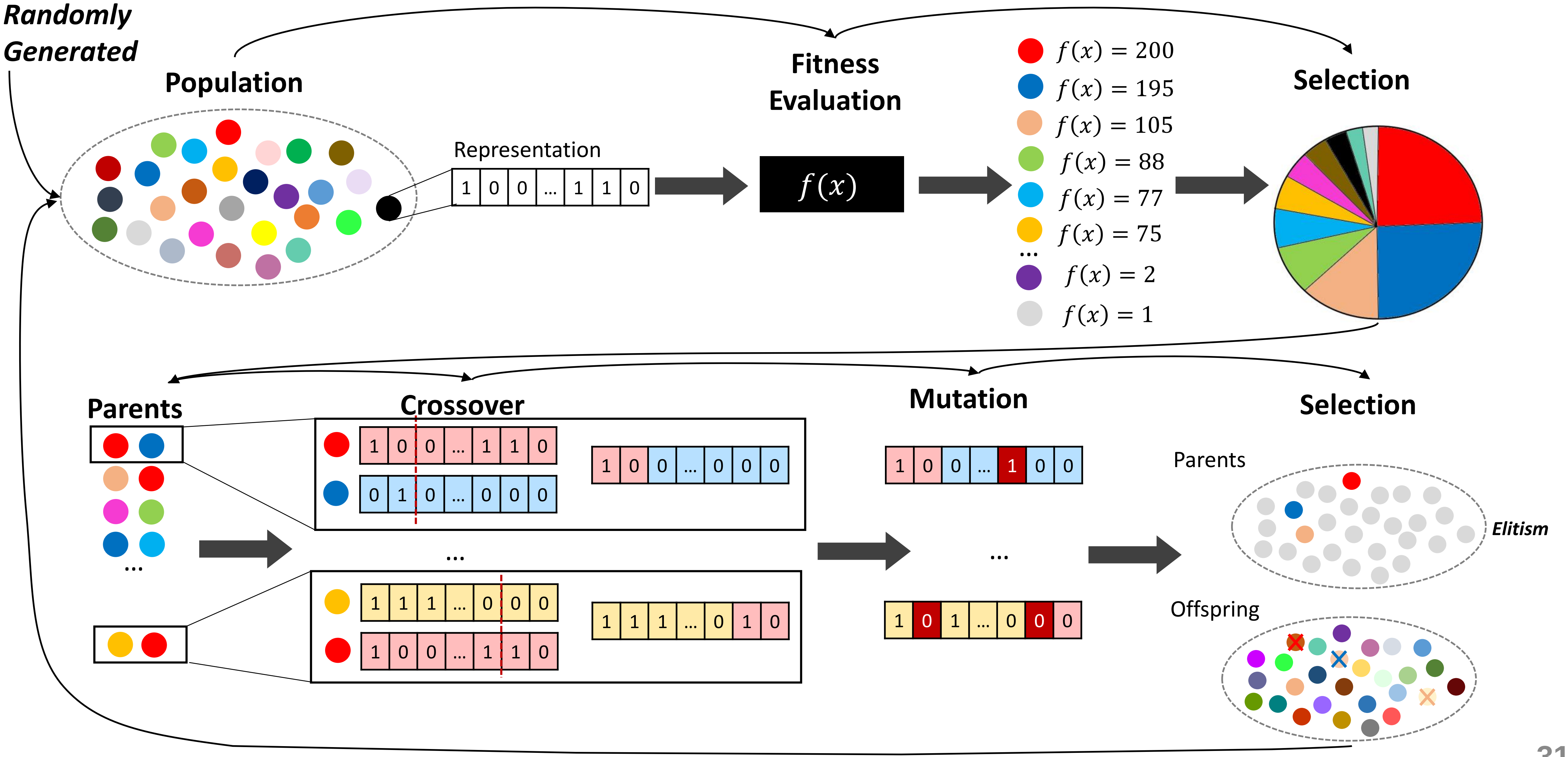
- Exercise 1: Propose and apply an Evolutionary Programming Algorithm (i.e. evolutionary algorithm with mutation operator only) for the problem.
  - Initial Population: uniformly randomly generated
  - Population Size: 10 to 50 solutions
  - Selection Strategy: roulette wheel selection
  - Reproduction Strategy: only mutation
  - Replacement strategy: generational replacement with elitism
- Exercise 2: Propose and apply a Genetic Algorithm (i.e. evolutionary algorithm with crossover operator and mutation operator) for the problem.
  - Initial Population: uniformly randomly generated
  - Population Size: 10 to 50 solutions
  - Selection Strategy: roulette wheel selection
  - Reproduction Strategy: crossover + mutation
  - Replacement strategy: generational replacement with elitism

Exercise 1

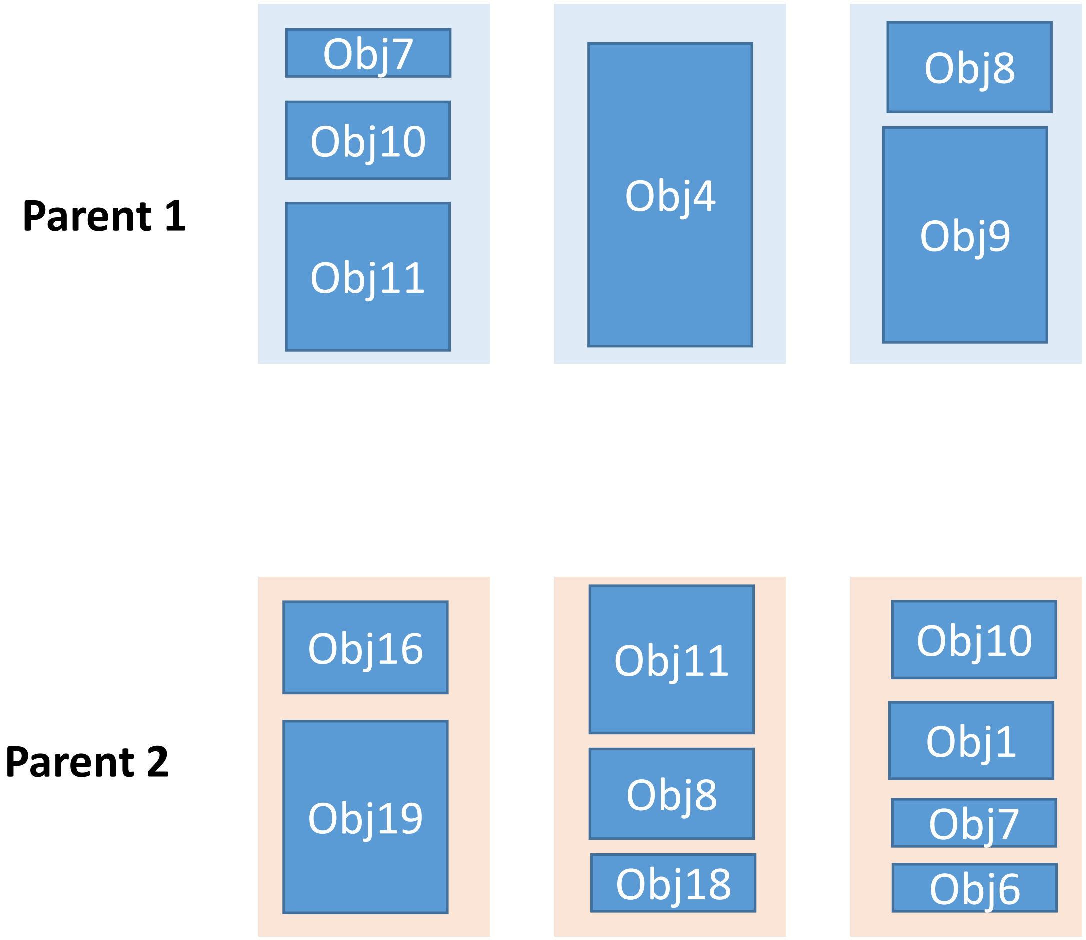


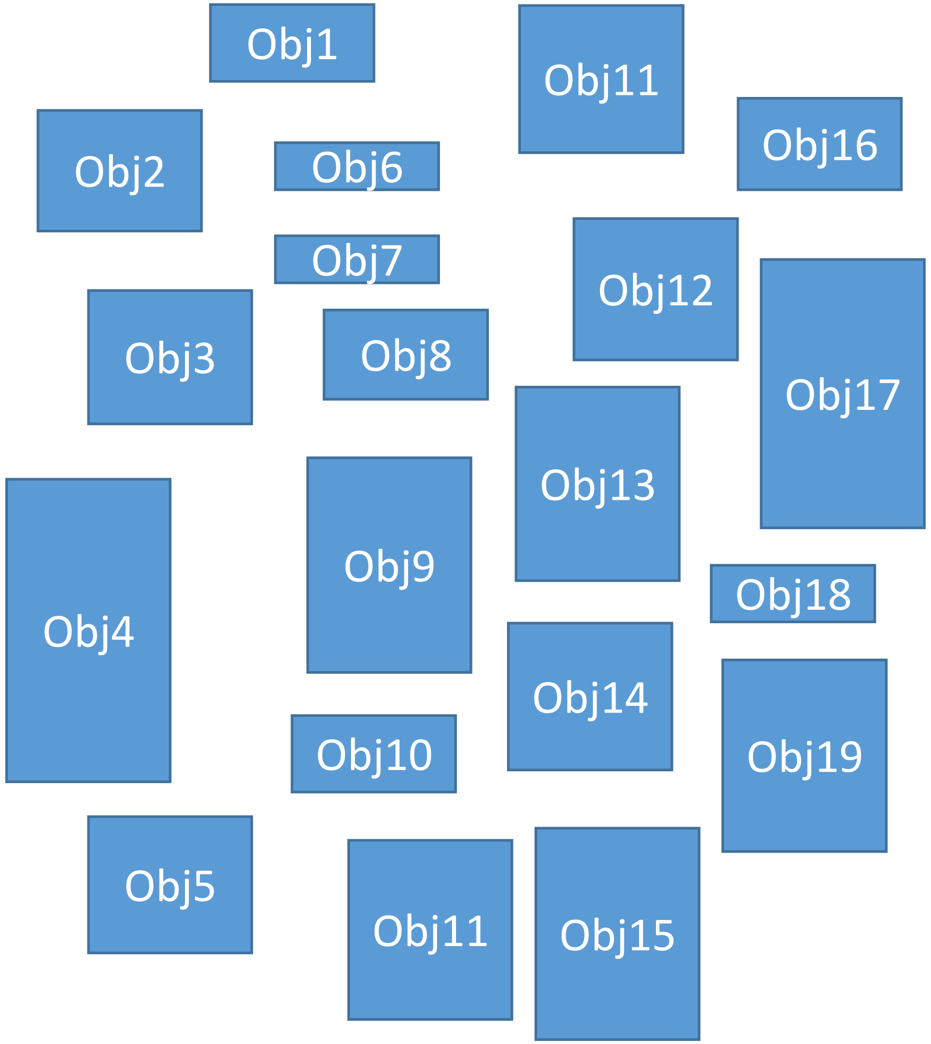
Randomly Generated
Population
Representation
1 0 0 … 1 1 0
Fitness Evaluation
$f(x)$
$f(x) = 200$
$f(x) = 195$
$f(x) = 105$
$f(x) = 88$
$f(x) = 77$
$f(x) = 75$
…
$f(x) = 2$
$f(x) = 1$
Selection
Parents
Mutation
1 0 0 … 1 1 0
…
1 1 1 … 0 0 0
Selection
Parents
Elitism
Offspring

Exercise 2

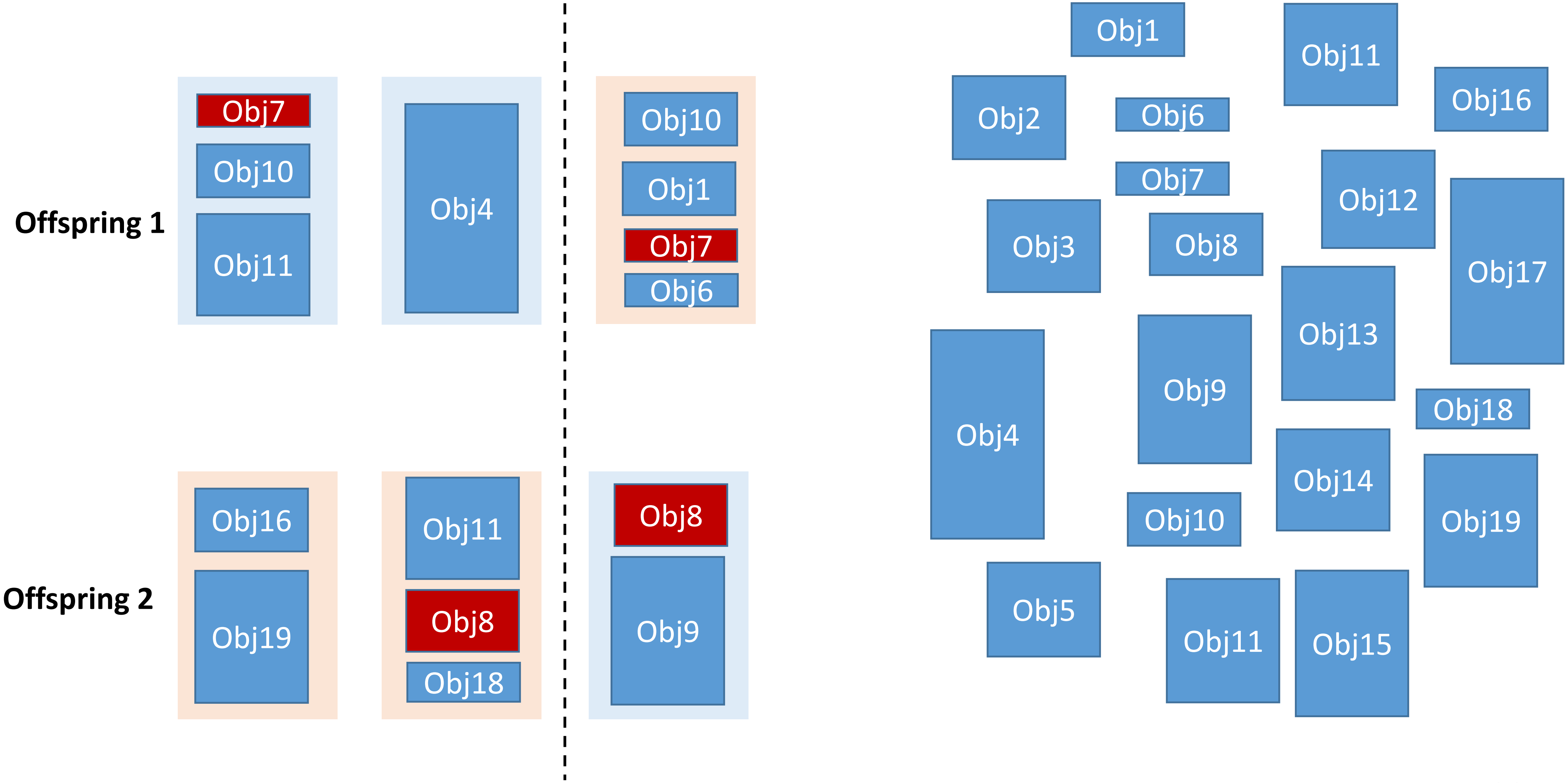

Randomly Generated
Population
Representation
1 0 0 … 1 1 0
Fitness Evaluation
$f(x)$
$f(x) = 200$
$f(x) = 195$
$f(x) = 105$
$f(x) = 88$
$f(x) = 77$
$f(x) = 75$
…
$f(x) = 2$
$f(x) = 1$
Selection
Parents
…
Crossover
1 0 0 … 1 1 0
0 1 0 … 0 0 0
1 0 0 … 0 0 0
…
1 1 1 … 0 0 0
1 0 0 … 1 1 0
1 1 1 … 0 1 0
Mutation
1 0 0 … 1 0 0
…
1 0 1 … 0 0 0
Selection
Parents
Elitism
Offspring

Crossover Operator

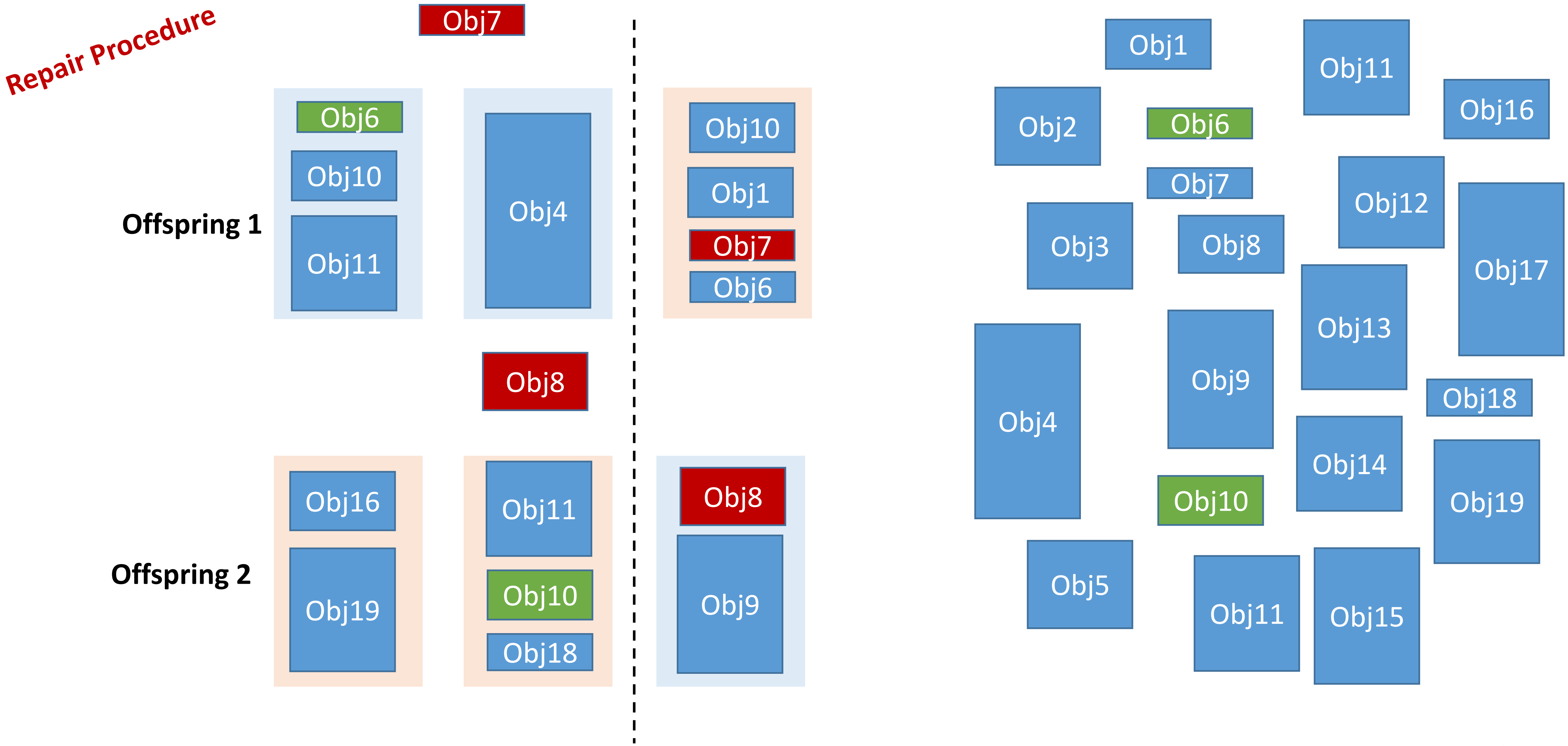




Crossover Operator



Crossover Operator



Types of Evolutionary Algorithms

Nuno Antunes Ribeiro

Assistant Professor

Engineering Systems and Design

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Types of Evolutionary Algorithms

Today's Class | Next Classes

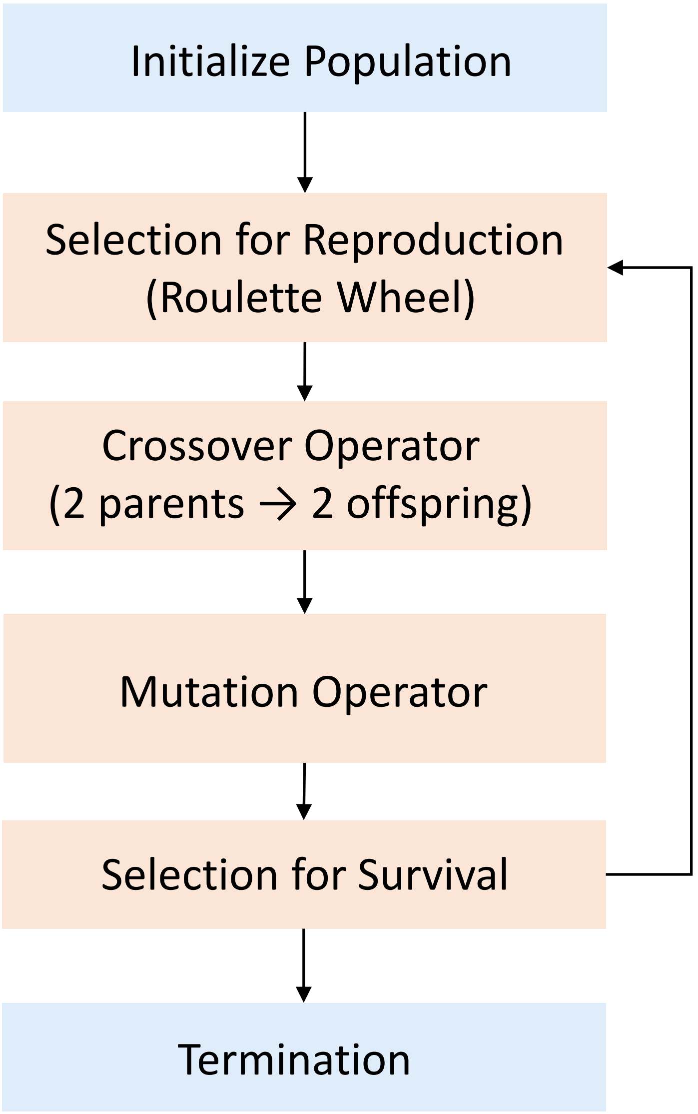
	Genetic Algorithms	Evolutionary Programming	Evolution Strategies	Differential Evolution	Genetic Programming	Neuroevolution
Year	~1975	**~1966**	**~1964**	~1997	~1992	~2002
Domain Structure	Vector of **real and discrete** variables	Vector of **real and discrete** variables	Vector of **real** variables	Vector of **real** variables	Tree-based **programs**	**Neural networks**
Reproduction	Crossover + Mutation	Only Mutation	(Recombination) + Mutation	Mutation + Crossover	Crossover + Mutation	Crossover + Mutation
Advantages	**General** approach to solve optimization problems of any type	Better at solving **constrained optimization** problems	Very effective on optimization problems with **continuous search space**	**Simple and fast**	**No analytical knowledge** is needed; This approach does scale with the problem size.	Powerful **integration** between **neural networks** and **genetic algorithms**
Drawbacks	It is **not a specialized technique**, thus performance can be worse for certain problems	Convergence to **local optima**	**Only** applicable to problems with **continuous** search spaces	Convergence to **local optima – only real** domains ca be considered	**Only** applicable to tree-based **programs** problems	**Only** applicable to artificial **neural networks**
Applications	**General applications**	**Constrained Opt. Problems** (e.g. scheduling ; routing ; designing systems)	**Continuous Optimization** Reinforcement Learning	Continuous Optimization **Control problems that require fast computation**	**Regression and Classification**; Robot navigation	**Reinforcement learning, Evolutionary Robotics, games**

Next Classes

- **Lecture 12** – Today's Class: **Evolution strategies** and **Differential Evolution**
- **Lecture 13** - Next Class: Using a genetic algorithm to **calibrate neural networks**
- **Lecture 14** - **Genetic Programming**: Evolutionary algorithm explores a program space rather than a solution space. GP is a form of program induction that allows to automatically generate programs that solve a given task
- **Lecture 15** – **Neuroevolution**: method for evolving artificial neural networks with a genetic algorithm
- **Lecture 15 and 16** – NSGA-II: Genetic Algorithm for **multi-objective optimization**

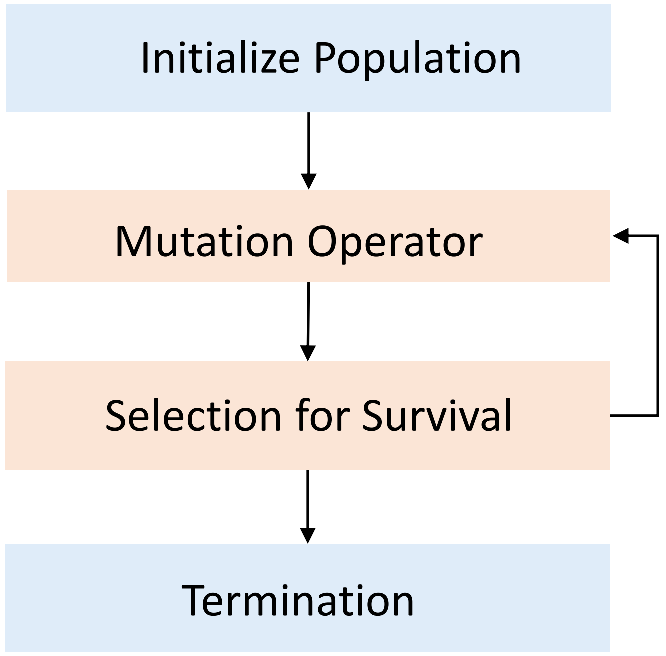
Genetic Algorithms

- Genetic algorithms have been developed by J. Holland in the 1970s (University of Michigan, USA) to understand the adaptive processes of natural systems. Then, they have been applied to optimization and machine learning in the 1980s.
- GAs are the most popular class of EAs. Traditionally, GAs are associated with the use of a binary representation but nowadays one can find GAs that use other types of representations.
- GA usually applies a crossover operator to two solutions that plays a major role, plus a mutation operator that randomly modifies the individual contents to promote diversity
- The replacement (survivor selection) is generational, that is, the parents are replaced systematically by the offsprings.



Evolutionary Programming

- First developed by J. Fogel in 1966, was one of the first genetic algorithms ever introduced.
- Evolutionary programming emphasizes on mutation and does not use crossover operators.
- Traditionally, the survivor selection process (replacement) is probabilistic and is based on a stochastic tournament selection.
- The framework of EP is less used than the other families of EA
- Contemporary EPs use self-adaptation principle of the parameters



EA for Continuous Optimization

- Randomized crossover operators for continuous problems might be very ineffective.
- Evolution Strategies and Differential Evolution are to Evolutionary Algorithms used to more effectively search for better solutions in a continuous search space
- **Typical crossover operator used in Genetic Algorithms**

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**

$\alpha P1_i + (1-\alpha) P2_i$ → α 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. $\alpha = 0.3$

Evolution Strategies	Differential Evolution
~1964	~1997
Vector of **real** variables	Vector of **real** variables
(Recombination) + Mutation	Mutation + Crossover
Very effective on optimization problems with **continuous search space**	**Simple and fast**
Only applicable to problems with **continuous** search spaces	Convergence to **local optima** – **only real** domains ca be considered
Continuous Optimization Reinforcement Learning	Continuous Optimization **Control problems that require fast computation**

Evolution Strategies

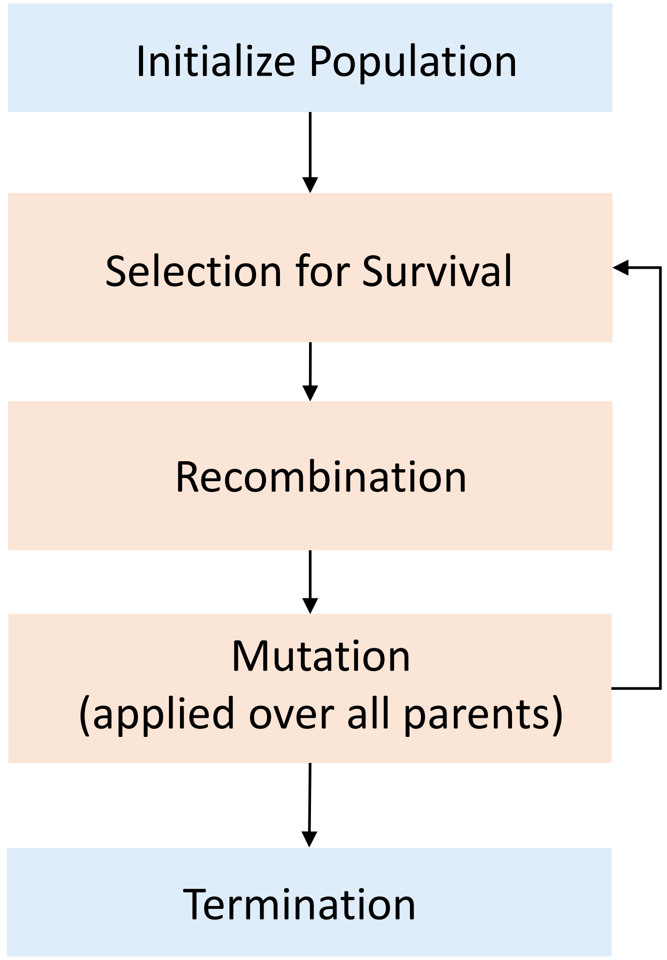
Nuno Antunes Ribeiro

Assistant Professor

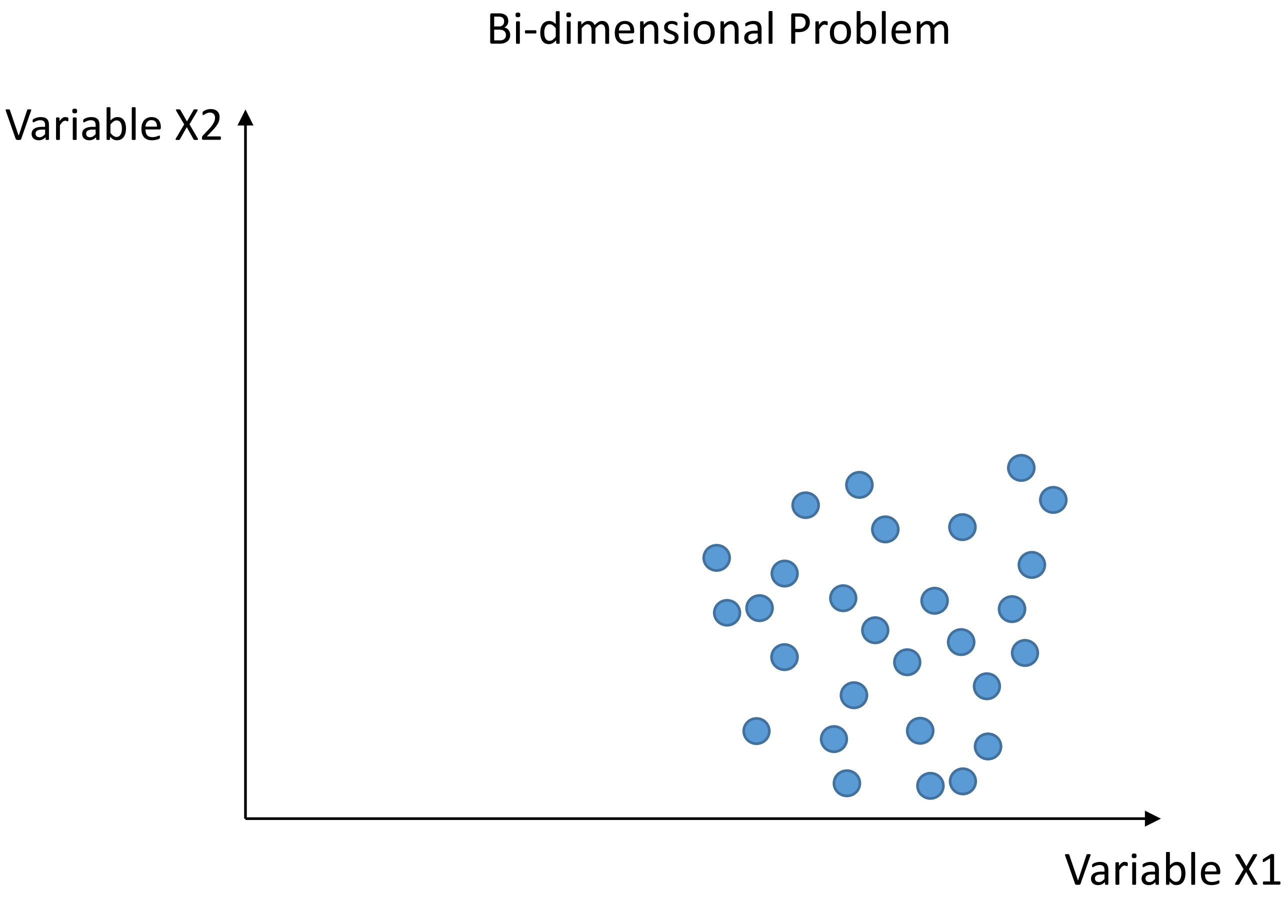
SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN | Engineering Systems and Design

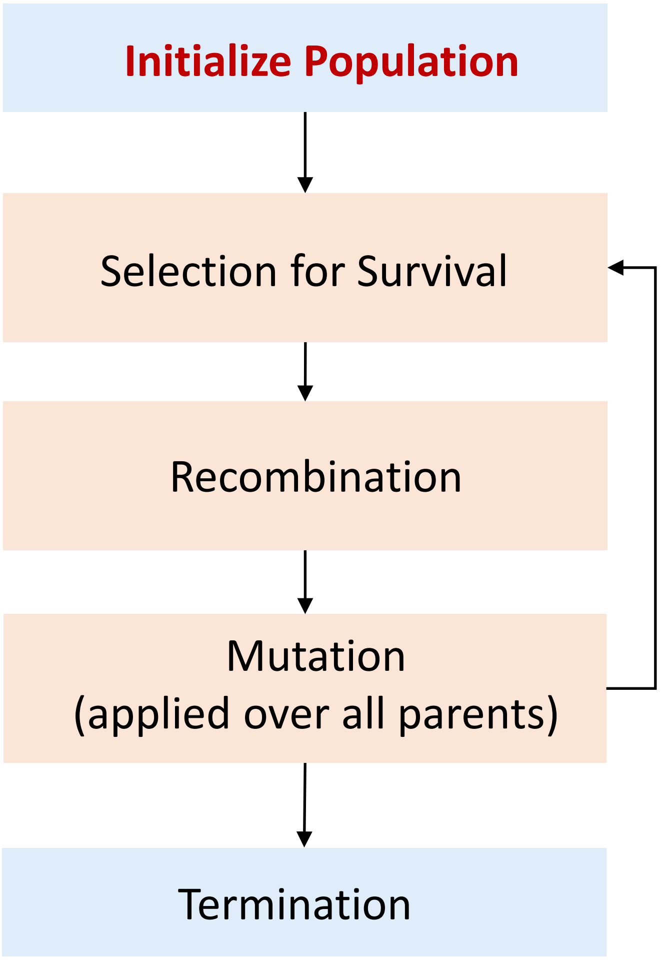
Evolution Strategies

- Developed by Rechenberg and Schwefel in Dortmund (Germany) in the 1960s - 10 years before Genetic Algorithms were used to solve mathematical functions by De Jong
- Evolutionary Algorithm for numerical optimization
- Search space: **vectors of real numbers**
- Different population treatments
- Recombination and Mutation as main search operations
- Idea: Self-adaptation of search – search operations automatically fine-tuned according to progress of search



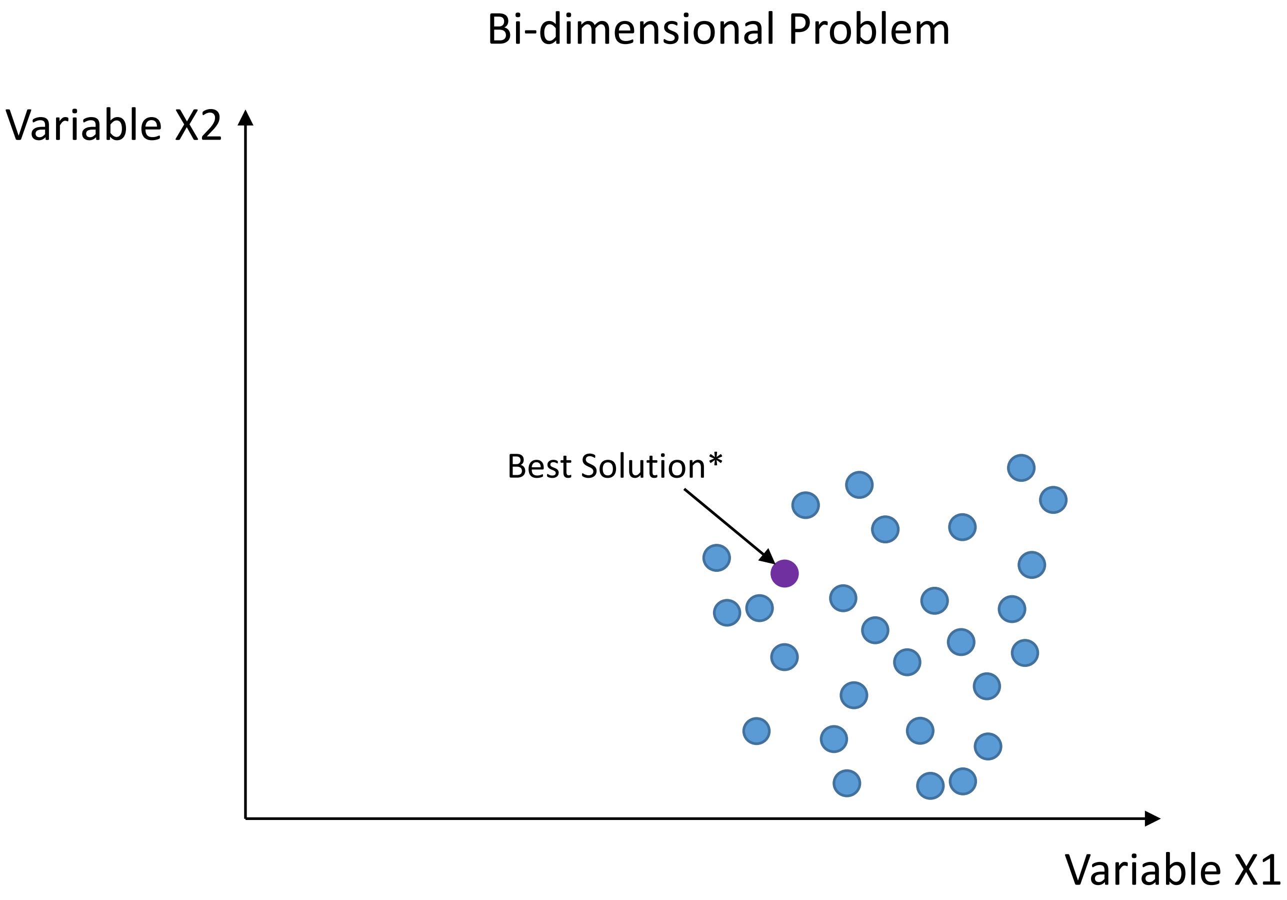
Evolution Strategies

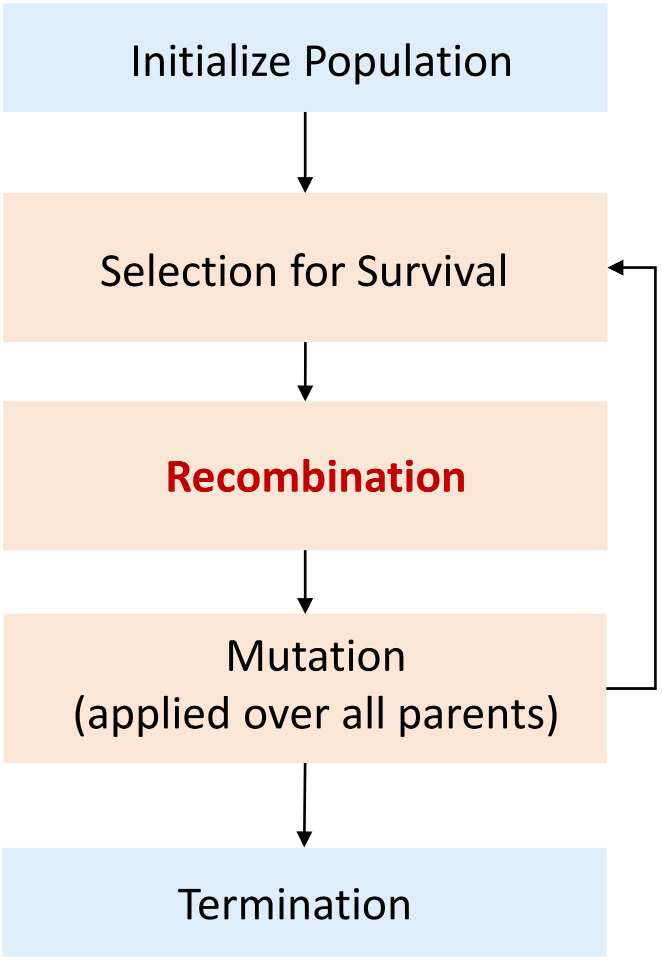




1. Sample a set of random solutions from a Normal distribution, with a mean $\mu = (\mu_{x1}, \mu_{x2})$ and standard deviation $\sigma = (\sigma_{x1}, \sigma_{x2})$

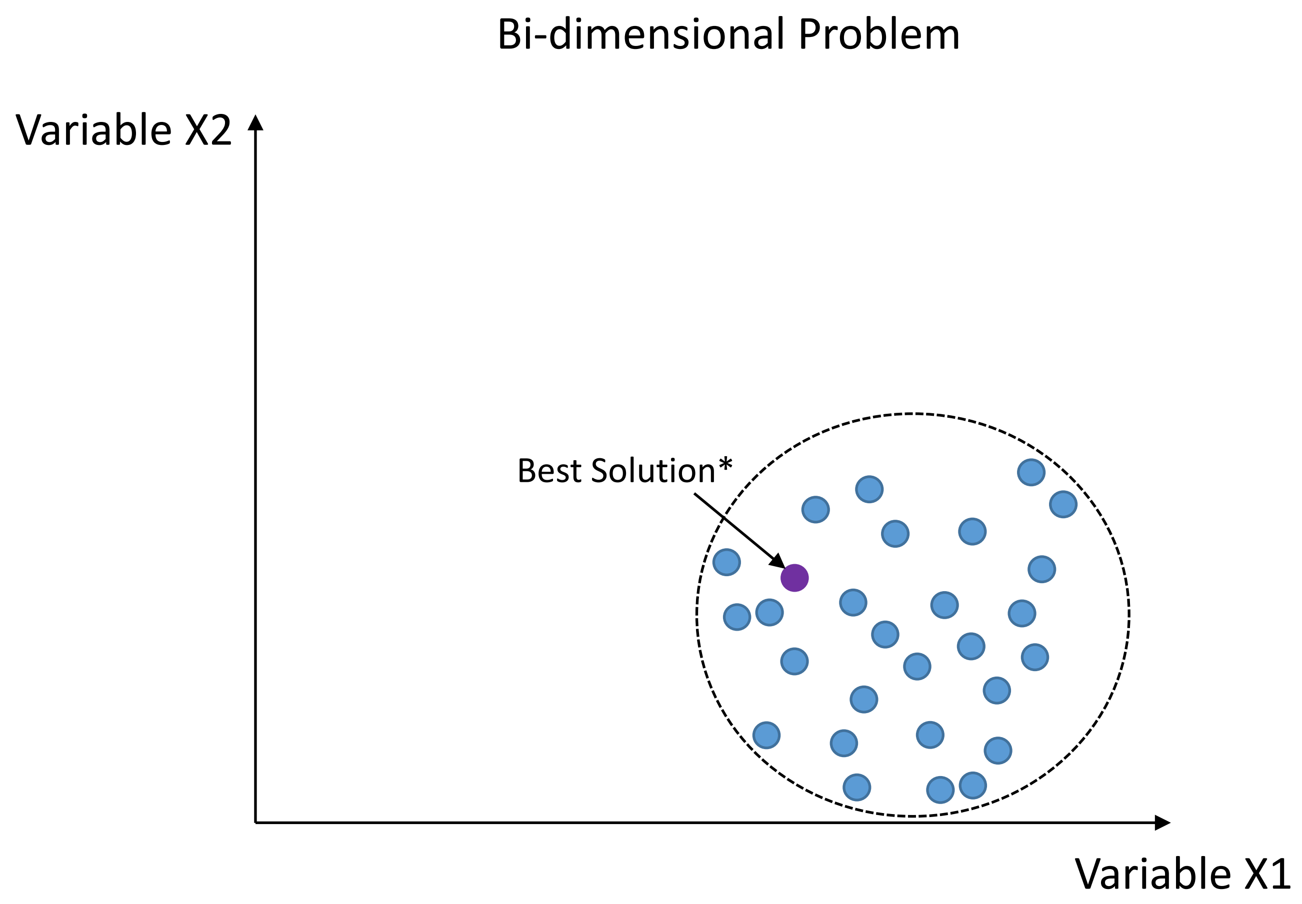
Evolution Strategies

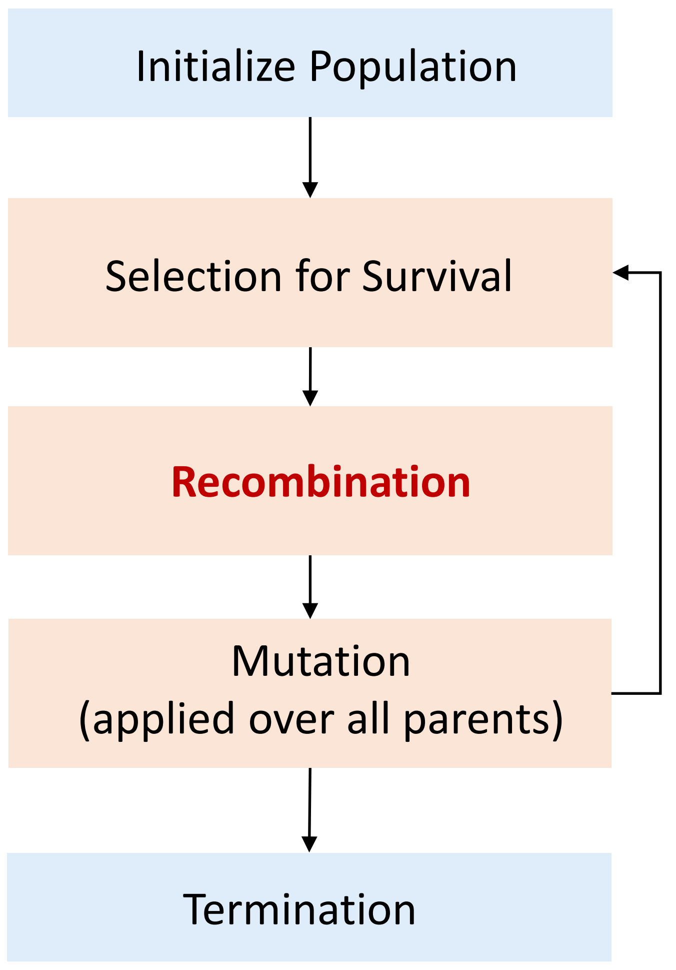




2. Select the best solution in the population $(X1^* \ ; \ X2^*)$
3. Set parameter $\mu_{x1} = X1^*; \mu_{x2} = X2^*$
4. Set parameter $\sigma_{x1} = std(X1) \ ; \sigma_{x2} = std(X2)$

Evolution Strategies

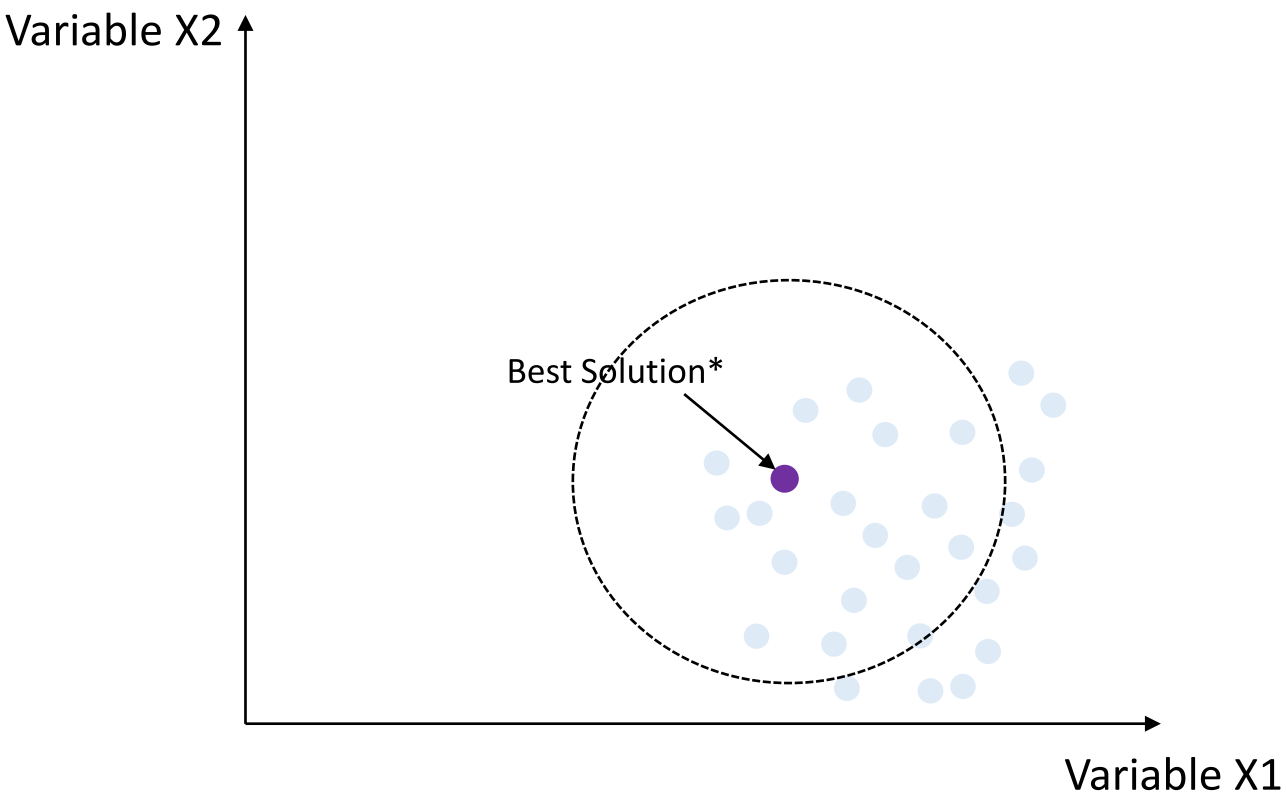


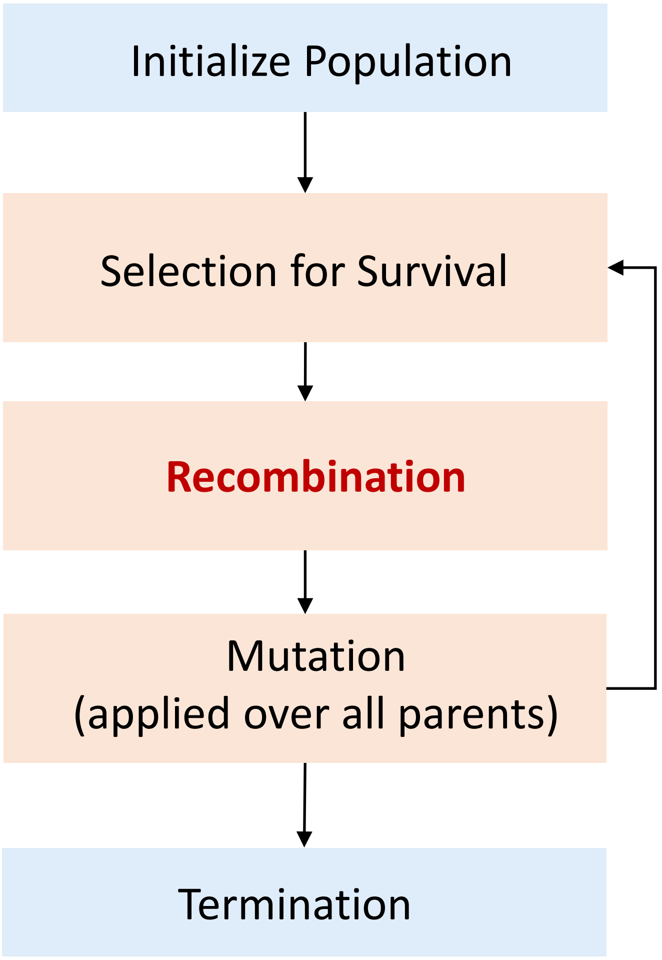


2. Select the best solution in the population $(X1^* \; ; \; X2^*)$
3. Set parameter $\mu_{x1} = X1^*; \mu_{x2} = X2^*$
4. Set parameter $\sigma_{x1} = std(X1) \; ; \sigma_{x2} = std(X2)$

Evolution Strategies

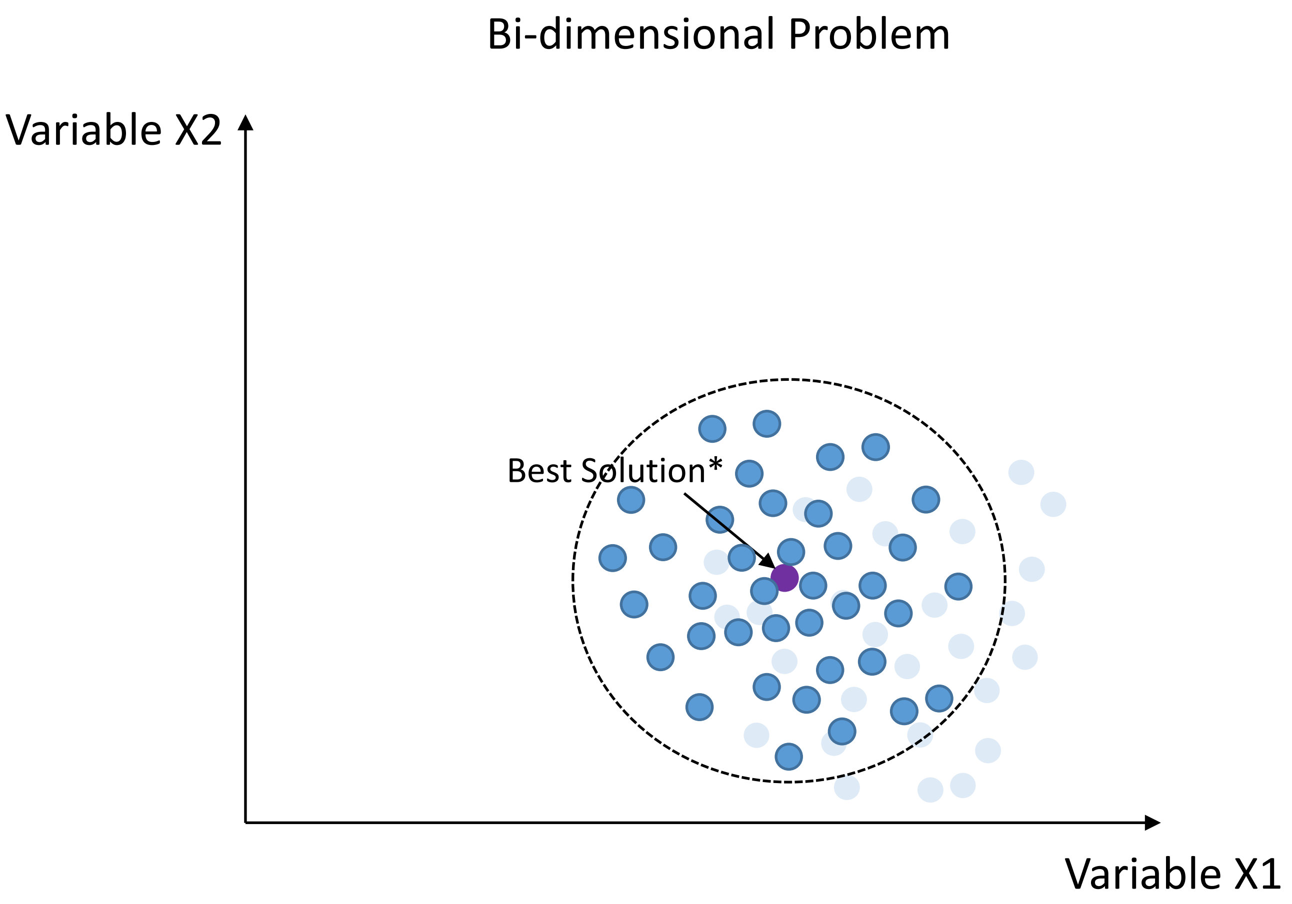
Bi-dimensional Problem

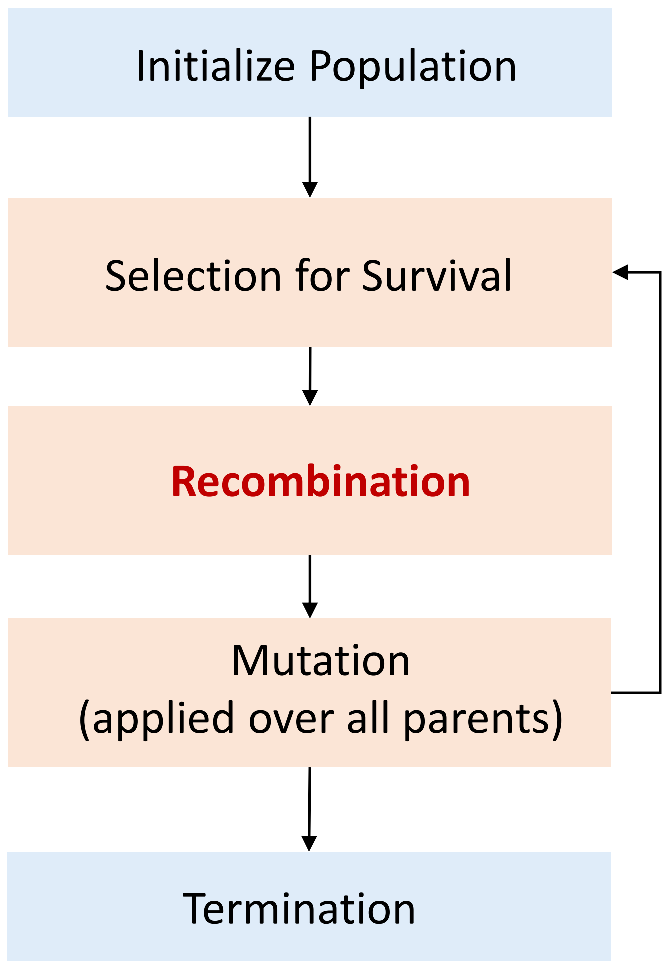




2. Select the best solution in the population $(X1^* \,;\, X2^*)$
3. Set parameter $\mu_{x1} = X1^*; \mu_{x2} = X2^*$
4. Set parameter $\sigma_{x1} = std(X1) \,; \sigma_{x2} = std(X2)$

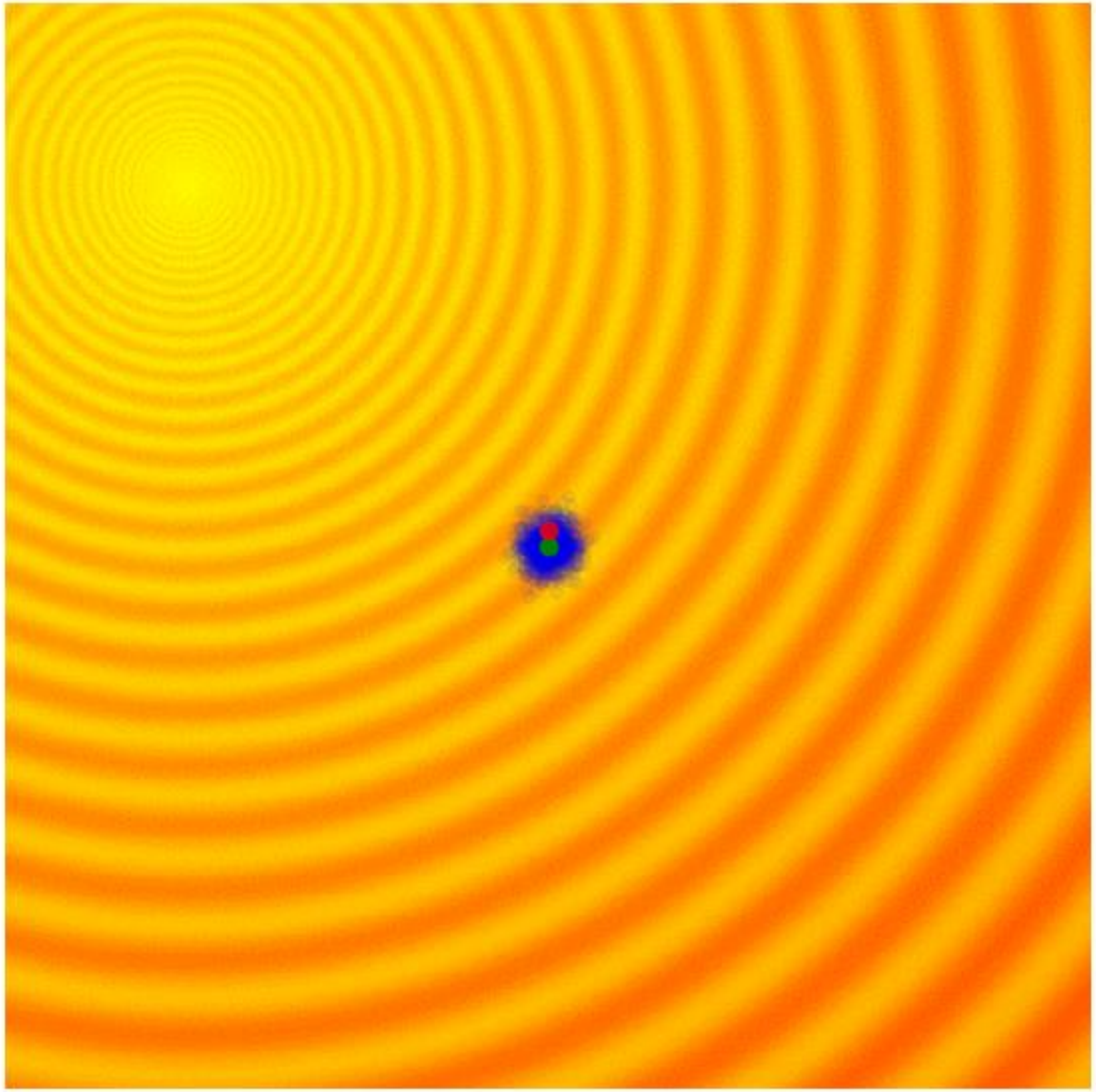
Evolution Strategies





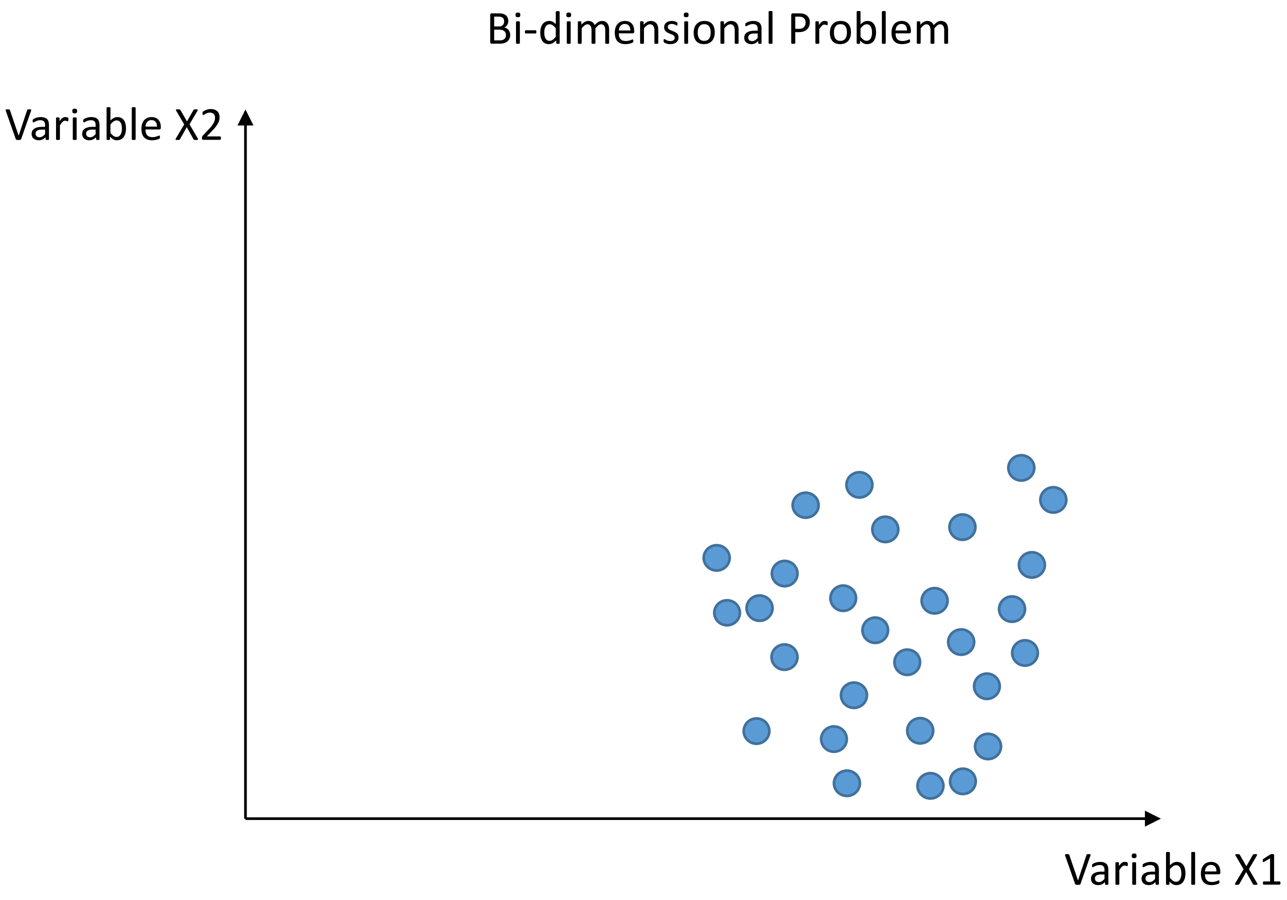
5. Sample a set of random solutions from a Normal distribution, with a mean $\mu = (\mu_{x1}, \mu_{x2})$ and standard deviation $\sigma = (\sigma_{x1}, \sigma_{x2})$

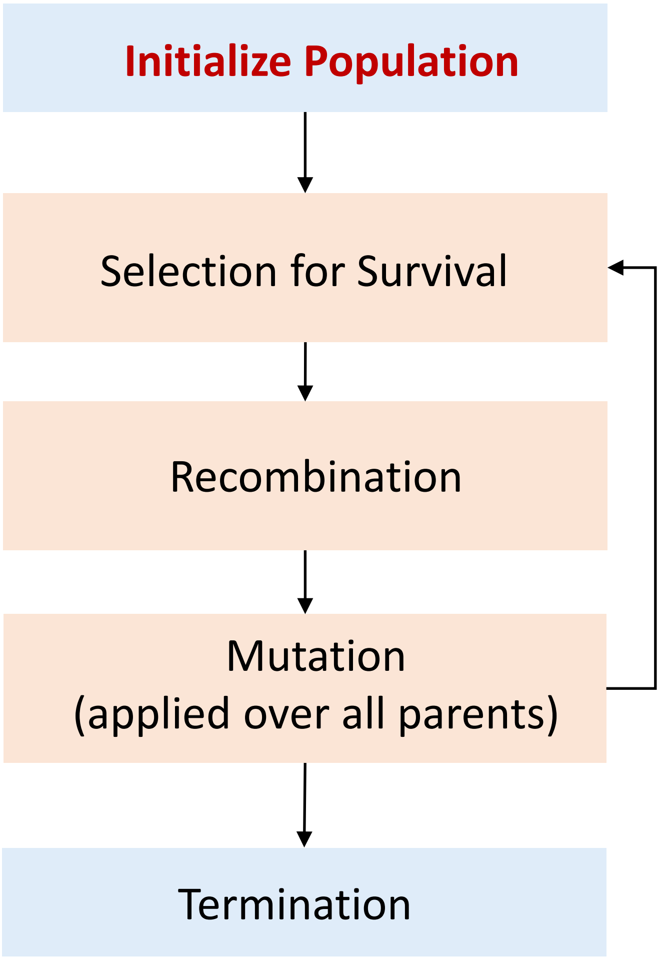
Evolution Strategies



A Visual Guide to Evolution Strategies | 大トロ (otoro.net)

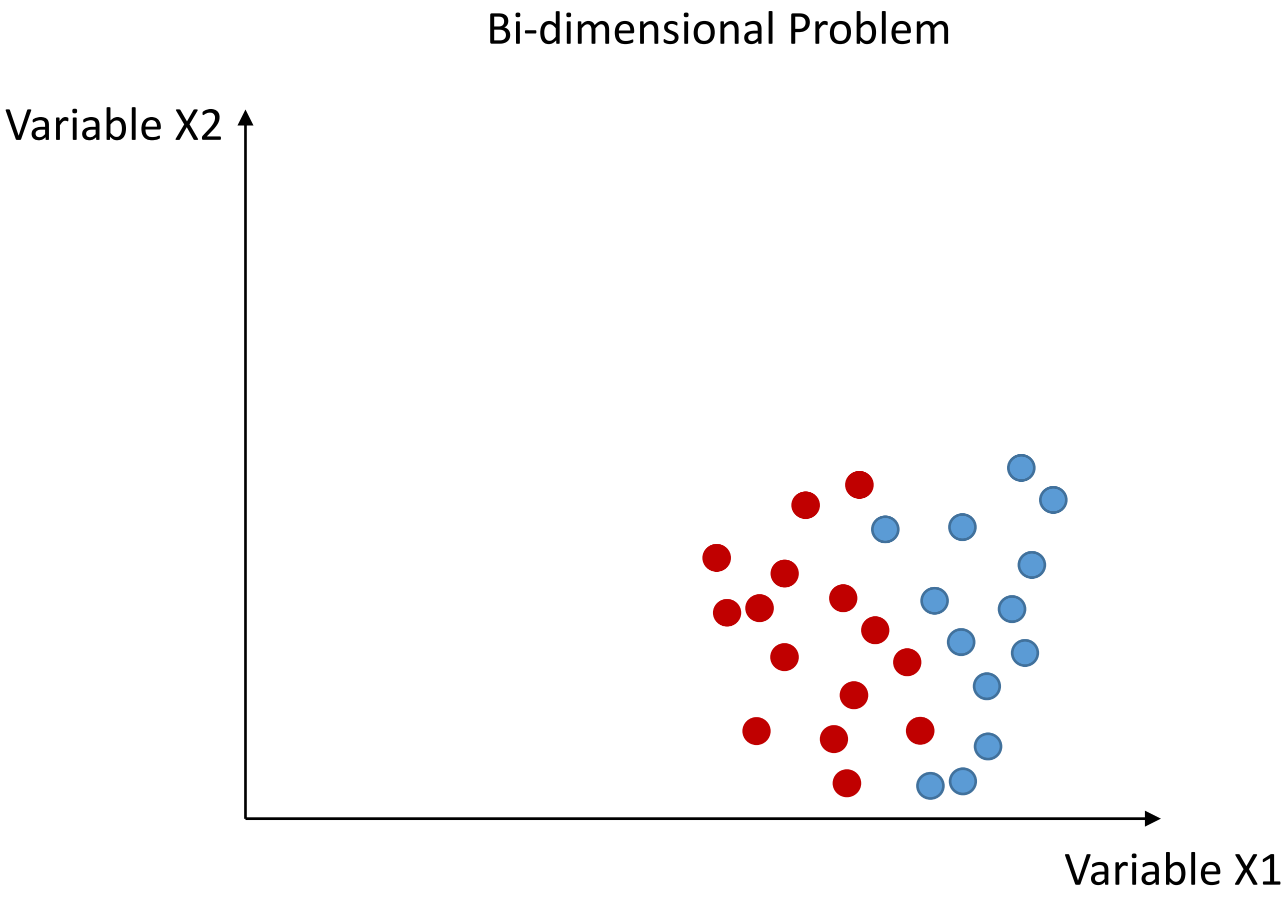
Evolution Strategies



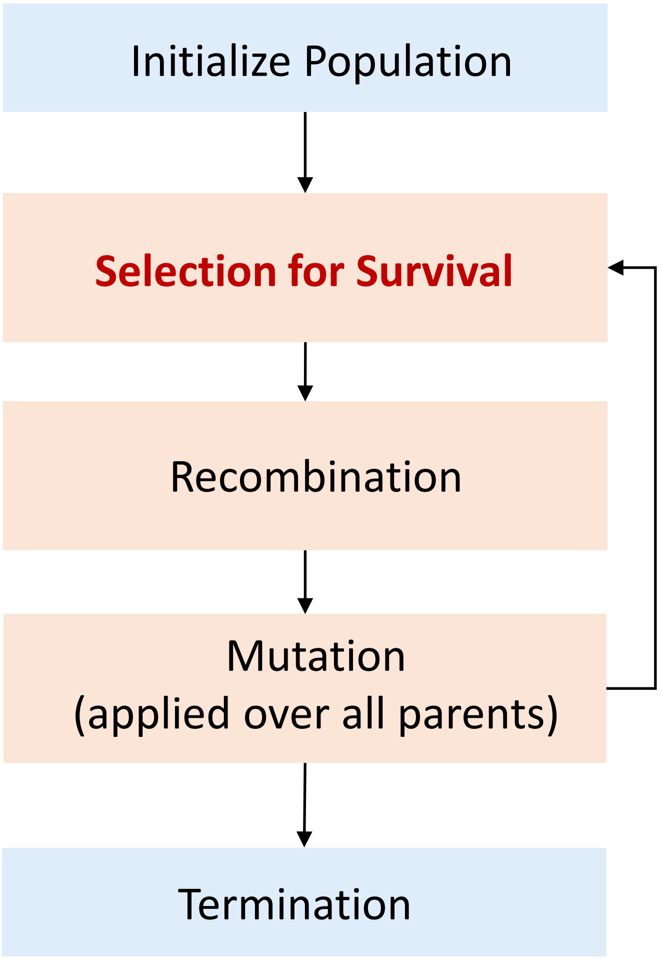


1. Sample a set of random solutions from a Normal distribution, with a mean $\mu = (\mu_{x1}, \mu_{x2})$ and standard deviation $\sigma = (\sigma_{x1}, \sigma_{x2})$

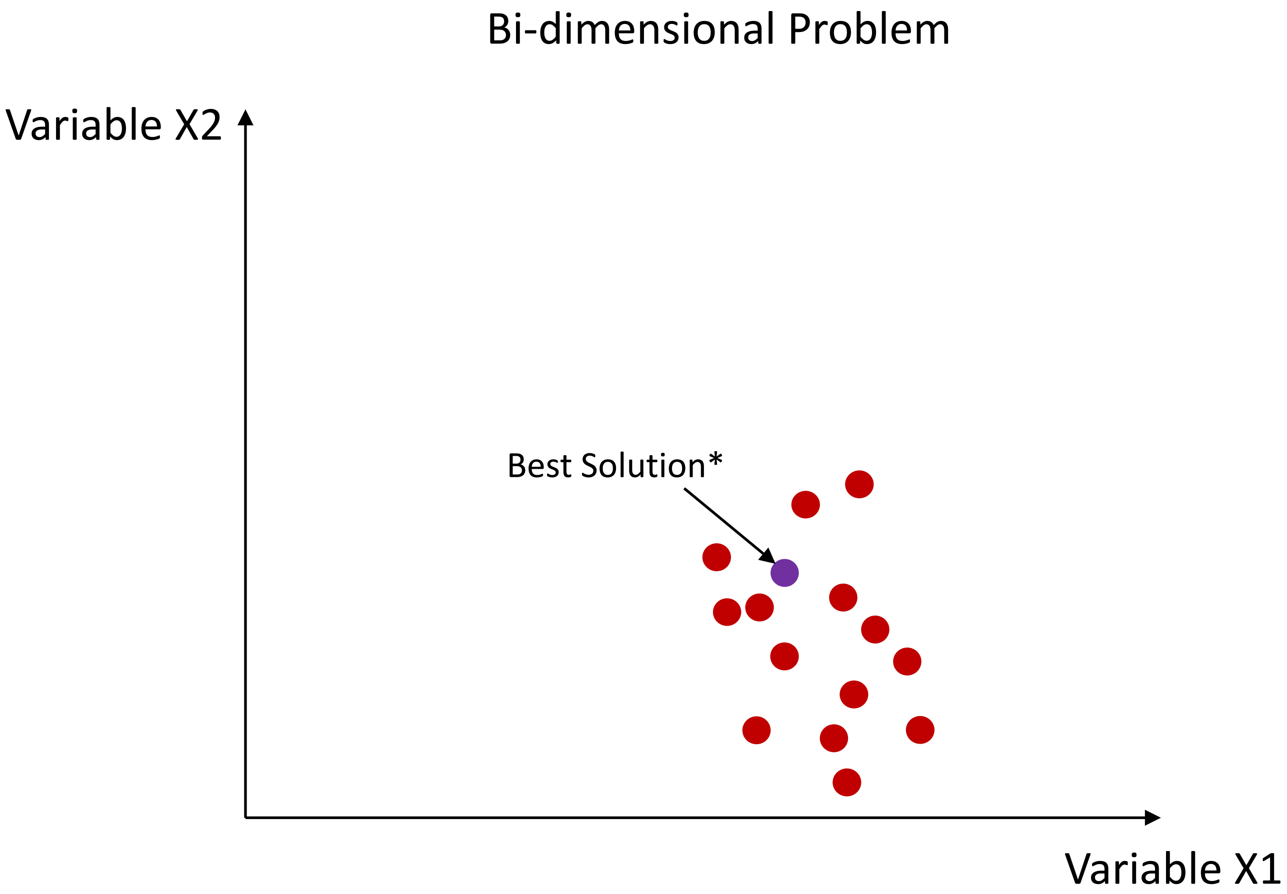
Evolution Strategies

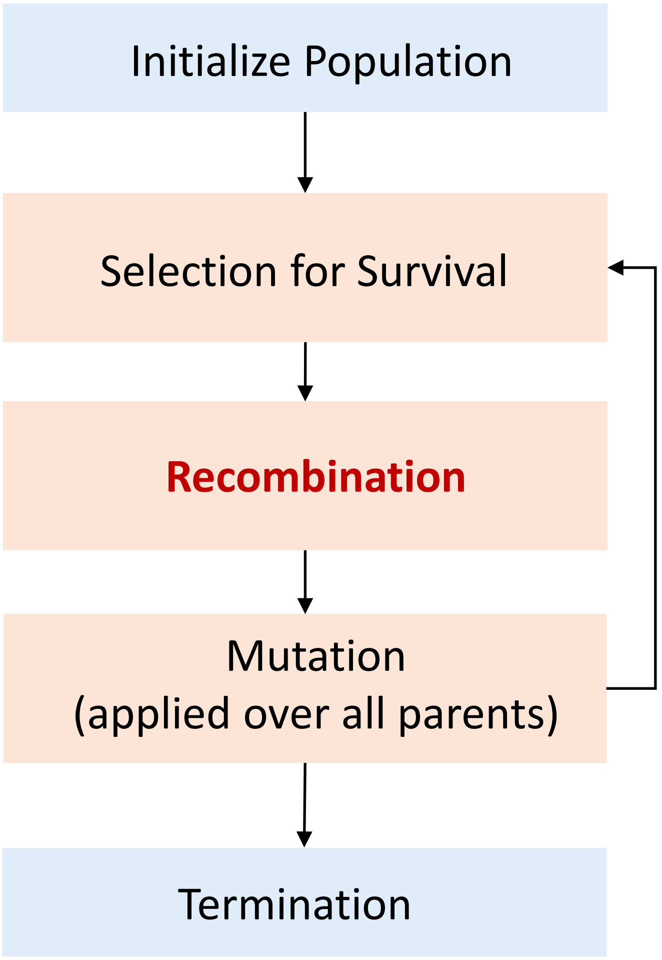


Elitism Selection



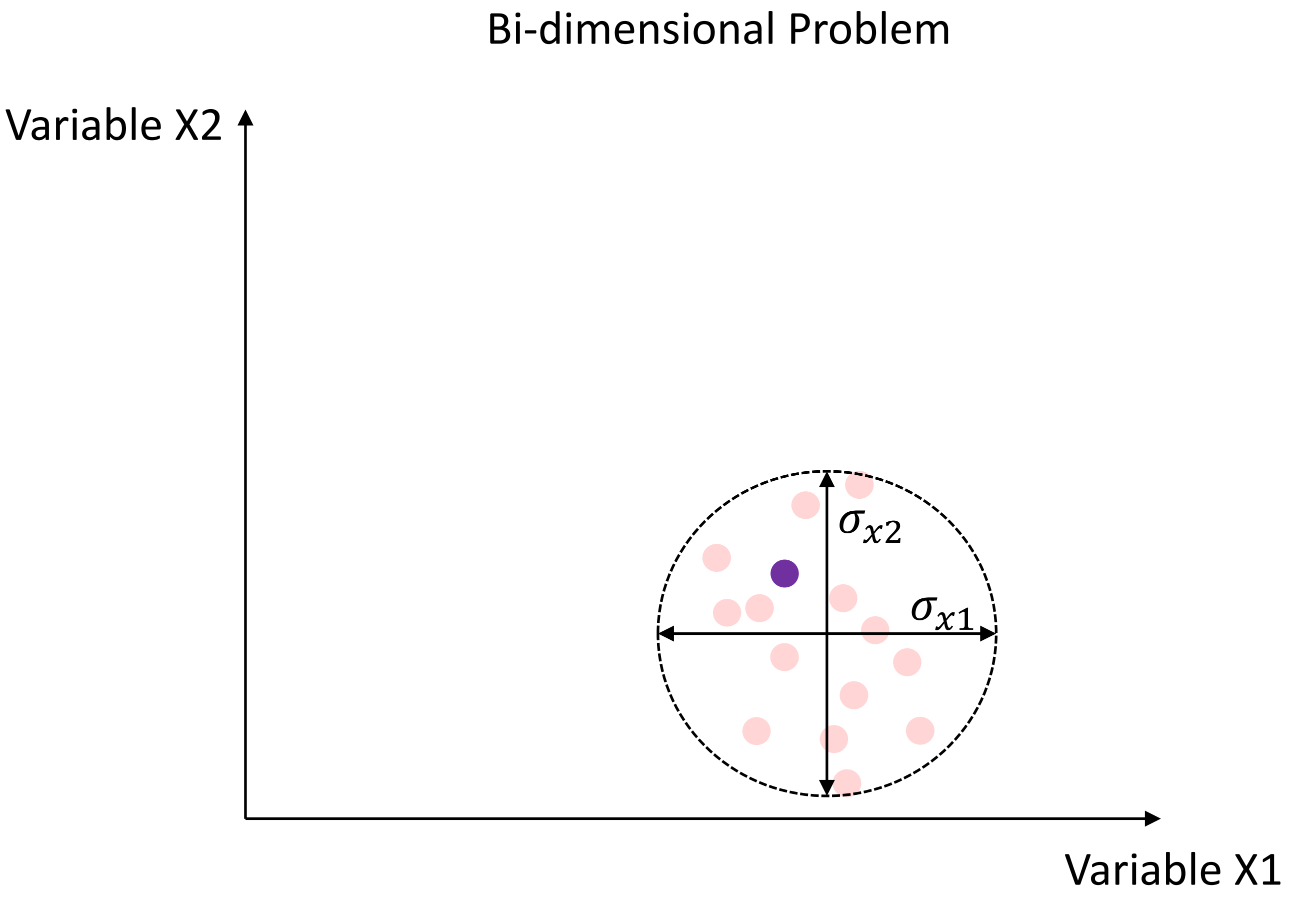
Evolution Strategies



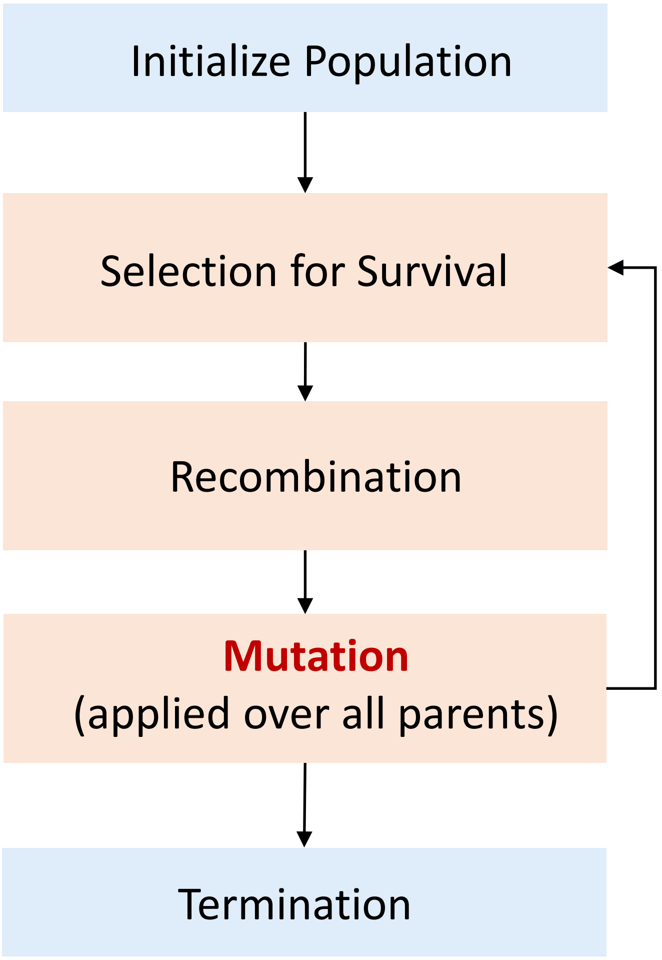


2. Select the best solution in the population $(X1^* \; ; \; X2^*)$
3. Set parameter $\mu_{x1} = X1^*; \mu_{x2} = X2^*$
4. Set parameter $\sigma_{x1} = std(X1) \; ; \sigma_{x2} = std(X2)$

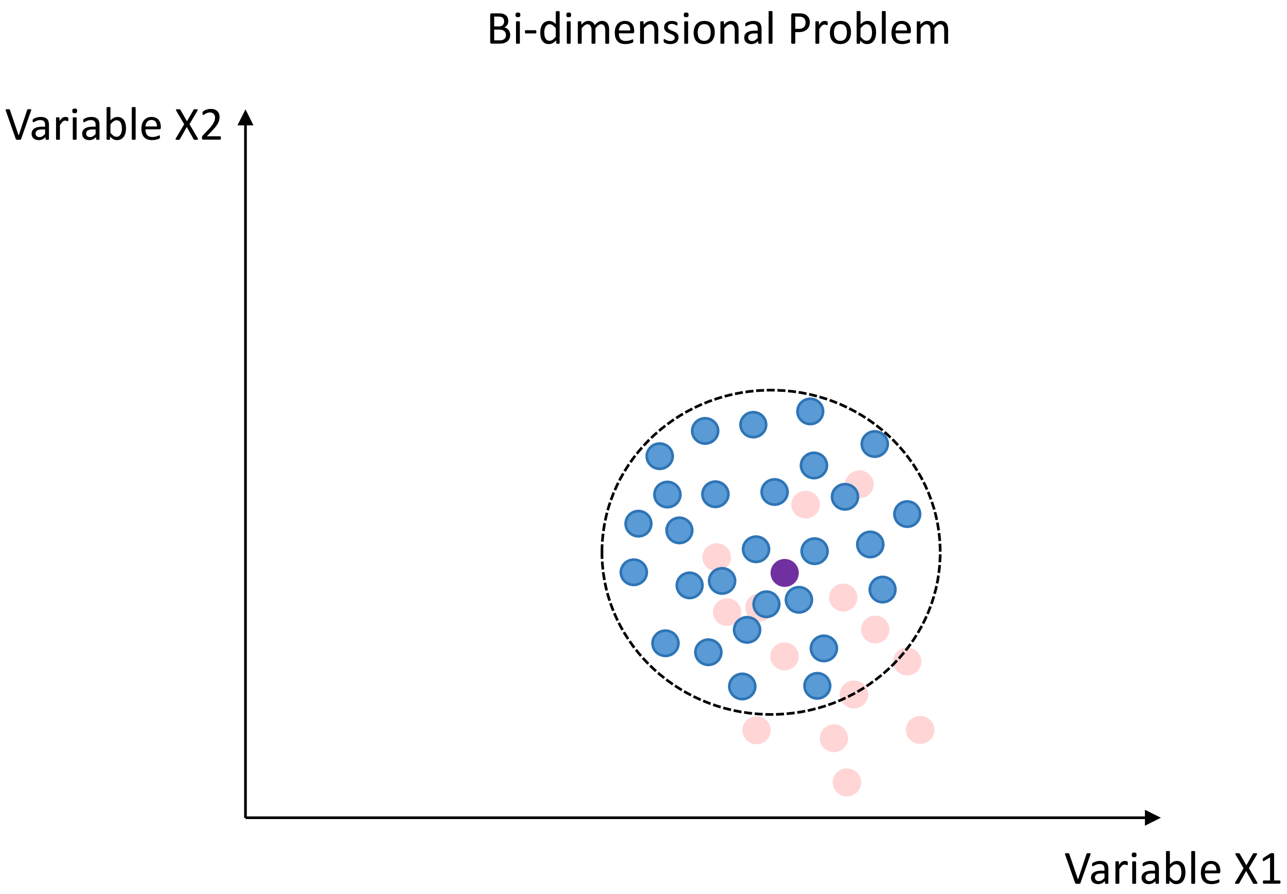
Evolution Strategies

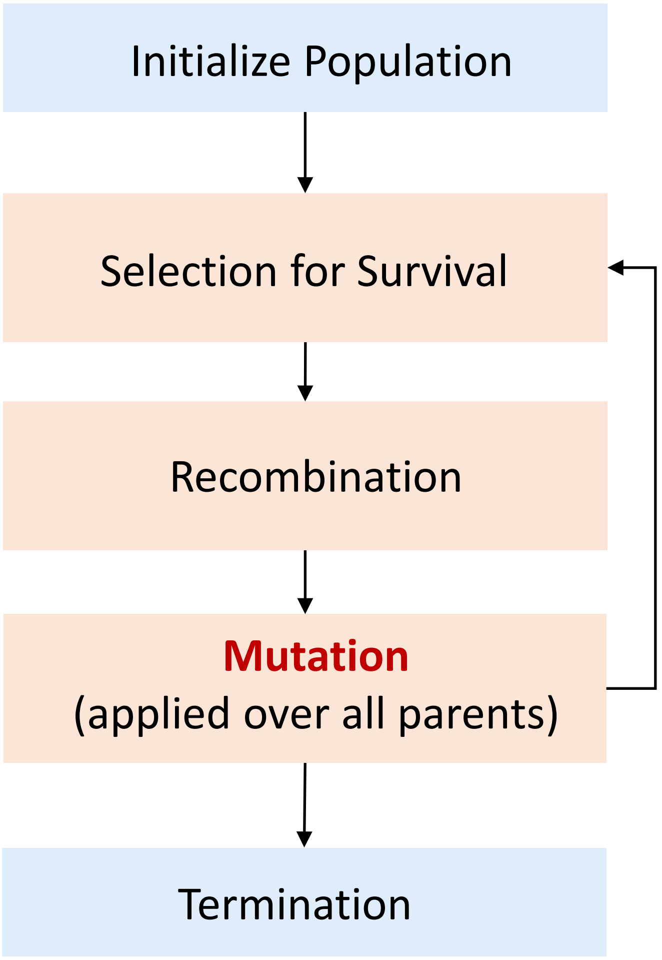


1. Sample a set of random solutions from a Normal distribution, with a mean $\mu = (\mu_{x1}, \mu_{x2})$ and standard deviation $\sigma = (\sigma_{x1}, \sigma_{x2})$



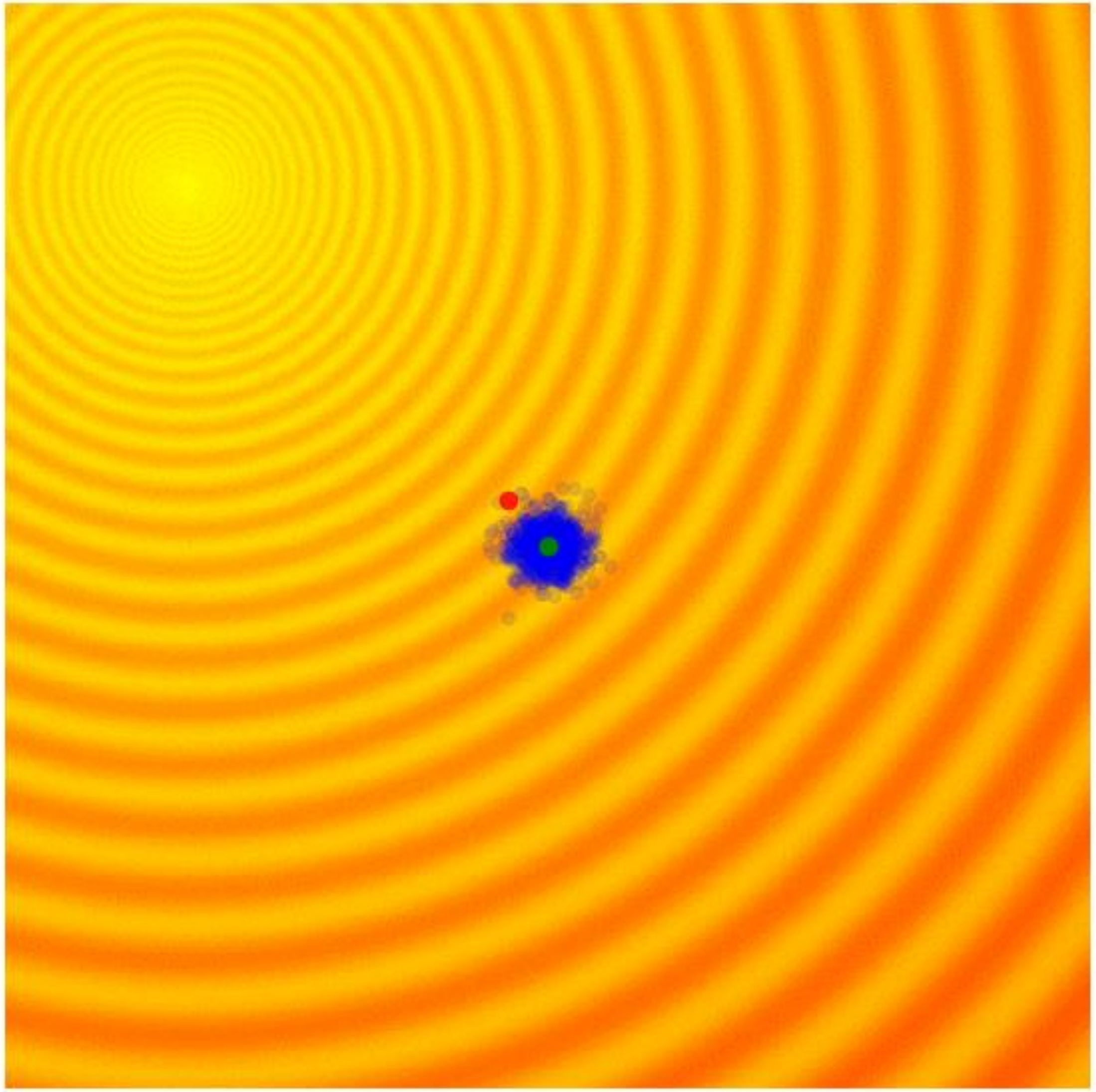
Evolution Strategies





1. Sample a set of random solutions from a Normal distribution, with a mean $\mu = (\mu_{x1}, \mu_{x2})$ and standard deviation $\sigma = (\sigma_{x1}, \sigma_{x2})$

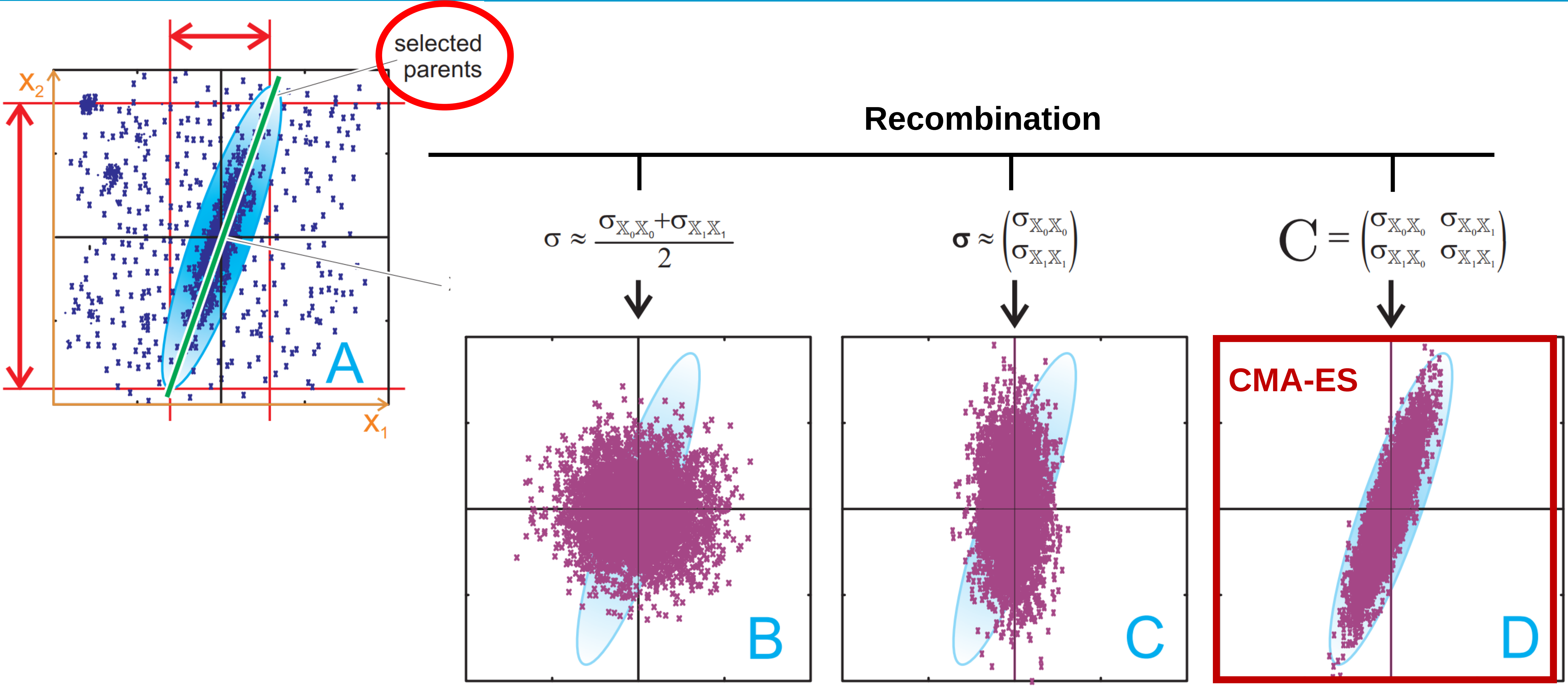
Evolution Strategies



The main drawback of CMA-Evolution Strategies is the performance if the number of model variables we need to solve is large, as the covariance calculation is time consuming

A Visual Guide to Evolution Strategies | 大トロ (otoro.net)

CMA-Evolution Strategies

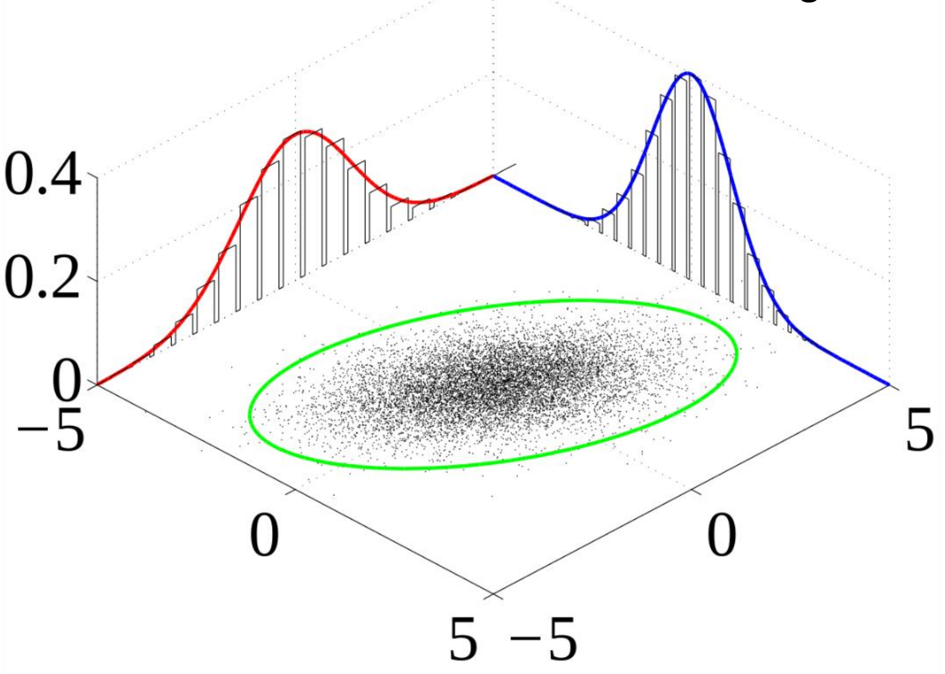


Mutation

CMA-Evolution Strategies

- The standard deviation σ accounts for the level of exploration: the larger σ the bigger search space we can sample our offspring population.
- In the previous example σ^{t+1} is highly correlated with σ^{t}, so the algorithm is not able to rapidly adjust the exploration space when needed (i.e. when the confidence level changes).
- CMA-ES, short for “Covariance Matrix Adaptation Evolution Strategy”, fixes the problem by tracking pairwise dependencies between the samples in the distribution with a covariance matrix C.

The main drawback of CMA-Evolution Strategies is the performance if the number of model variables we need to solve is large, as the covariance calculation is time consuming



https://janakiev.com/blog/covariance-matrix/

Differential Evolution

Nuno Antunes Ribeiro

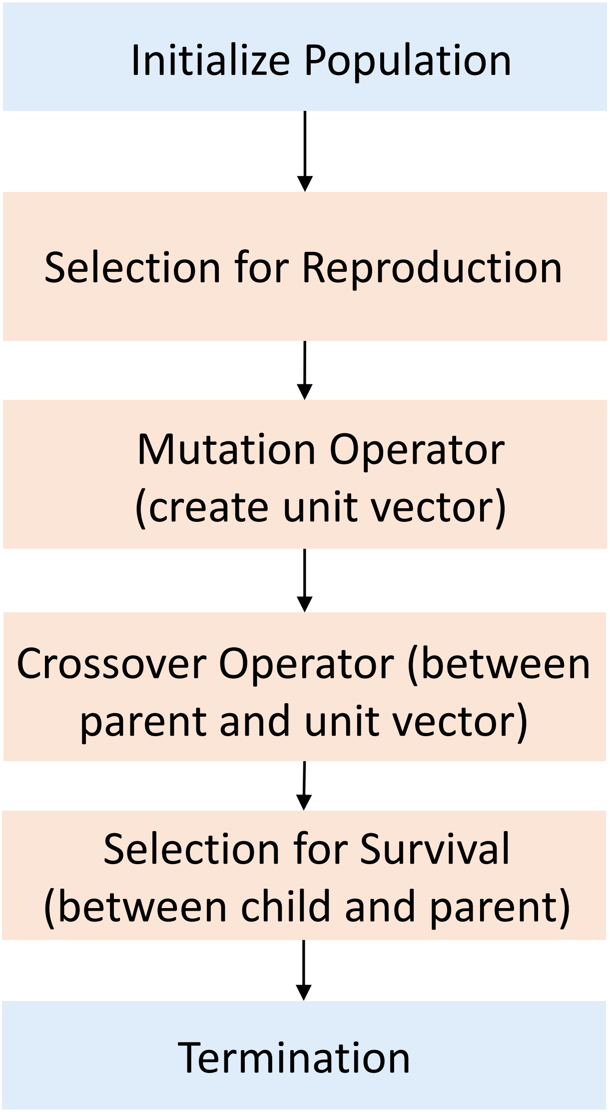
Assistant Professor

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Engineering Systems and Design

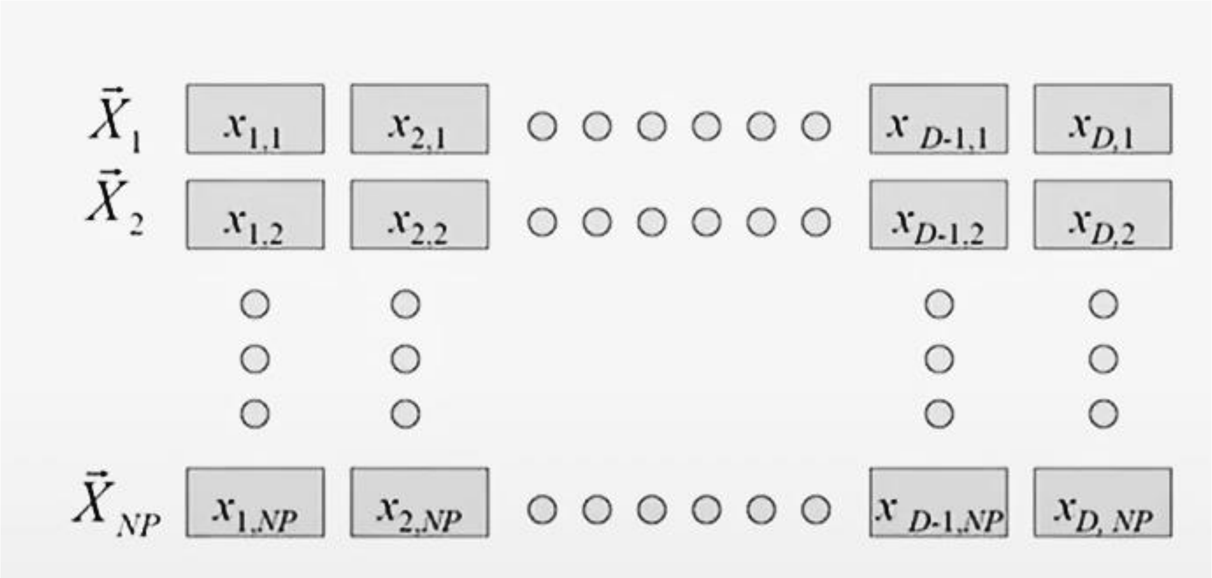
Differential Evolution

- Developed by Storn and Price in the mid-1990s.
- The DE algorithm is advantageous over the other approaches because it requires very few control parameters.
- Differential Evolution differs from standard genetic algorithms since it relies upon **distance and directional information** through unit vectors for reproduction.
- Another peculiar characteristic is that **crossover is applied after mutation**
- In the selection for survival each **offspring competes with its direct parent**, replacing it only if it has better objective values
- Complexity is very low as compared to some of the most competitive optimizers like CMA-ES



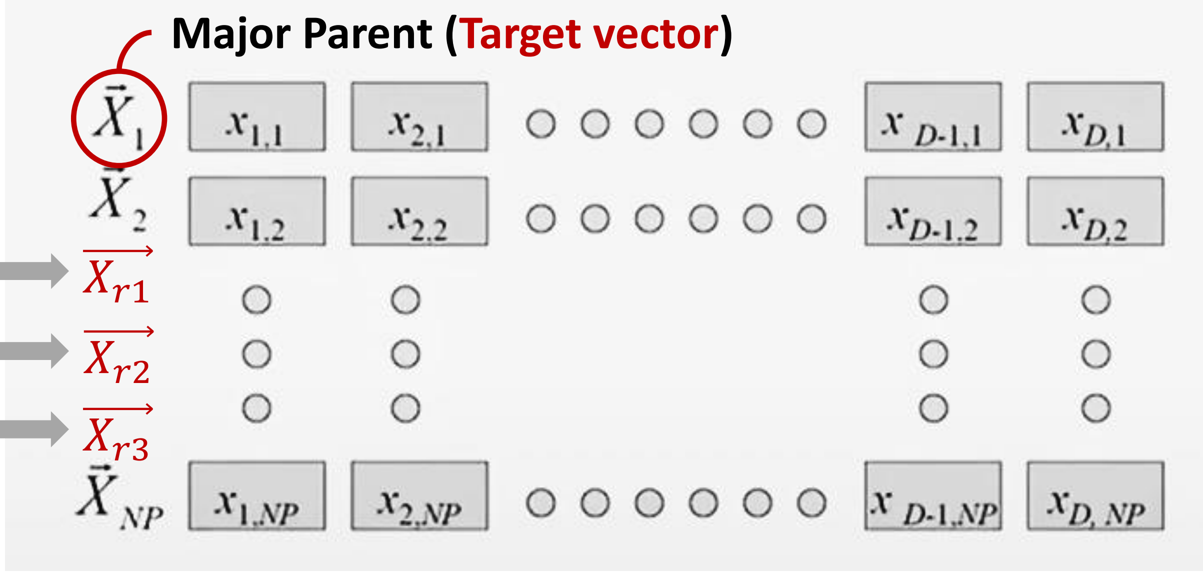
Differential Evolution

- Let's consider an optimization problem with D decision variables.
- We initialize population through randomization of NP solutions.
- Each solution X is a vector of decision variables



Mutation Operator in DE

- For each parent randomly select 3 other parents for mutation
- Add the **weighted difference of two of the parent vectors** to the third parent vector to form a donor vector V_{iG}, where G is the index of the major parent



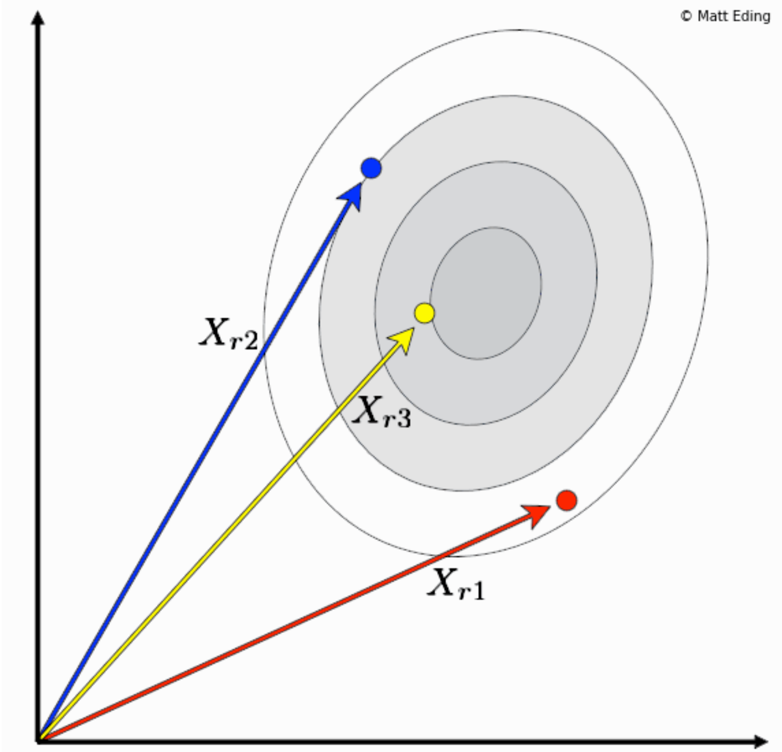
Donor Parents

Selected Randomly

Donor vector | Unit vector | Difference vector

$$V_{iG} = X_{r1_iG} + \lambda(X_{r1_iG} - X_{r1_iG})$$

λ is a factor from 0 to 2

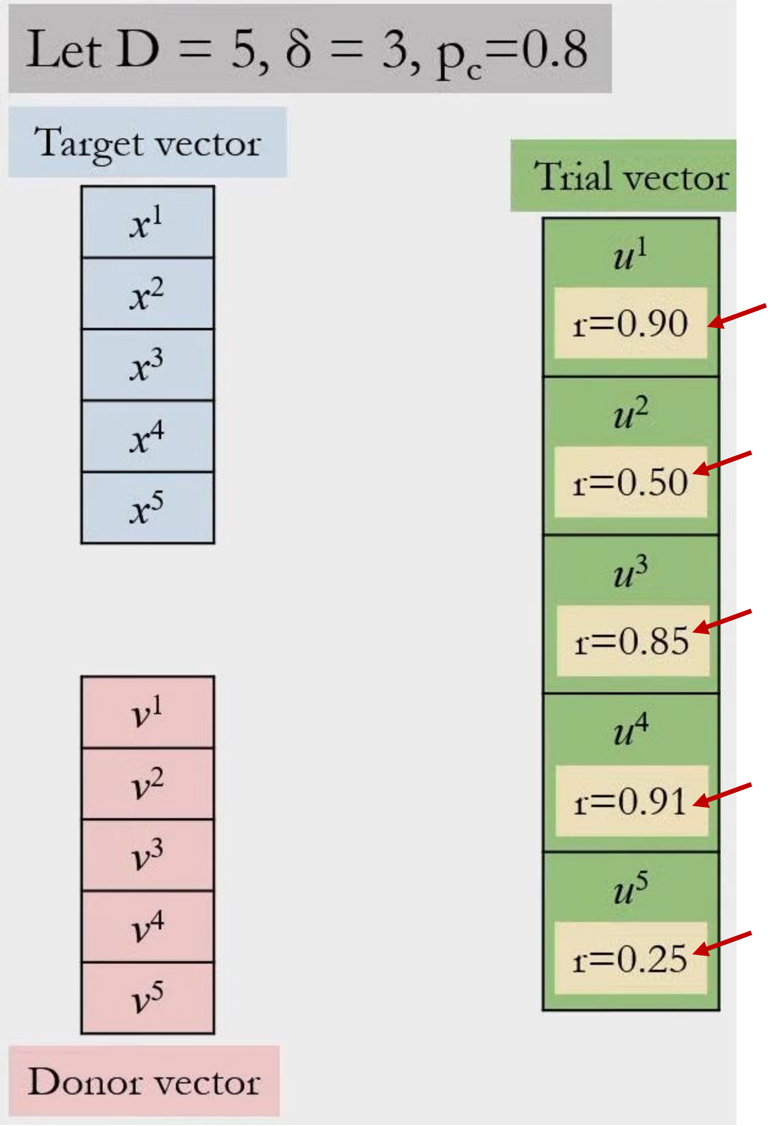


https://matteding.github.io/2019/04/17/differential-evolution/

Crossover Operator in DE

$$u^j = \begin{cases} v^j & \text{if } r \le p_c \text{ OR } j = \delta \\ x^j & \text{if } r > p_c \text{ AND } j \ne \delta \end{cases}$$

- Binomial Crossover
  - Performed to increase diversity
  - 3 vectors
    - x_i target vector
    - v_i donor vector
    - u_i trial vector (<u>what we want to compute</u>)
  - 3 parameters:
    - <u>r_i random number between 0 and 1</u>
    - p_c crossover probability (selected by the user)
    - δ randomly selected variable location $\delta \in \{1,2,3, \ldots D\}$

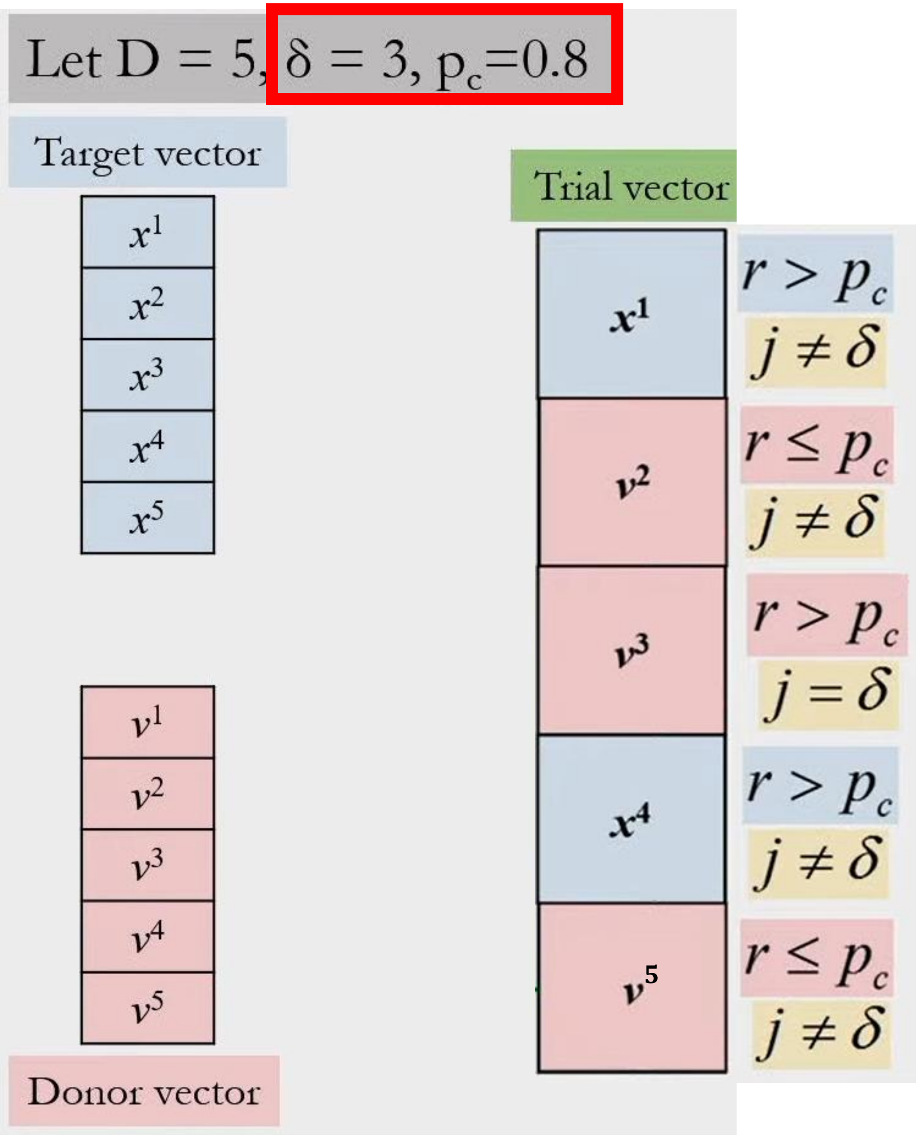


Crossover Operator in DE

$$u^j = \begin{cases} v^j & \text{if } r \le p_c \text{ OR } j = \delta \\ x^j & \text{if } r > p_c \text{ AND } j \ne \delta \end{cases}$$

- Binomial Crossover
  - Performed to increase diversity
  - 3 vectors
    - x_i target vector
    - v_i donor vector
    - u_i trial vector (what we want to compute)
  - 3 parameters:
    - r_i random number between 0 and 1
    - <u>p_c crossover probability (selected by the user)</u>
    - <u>δ randomly selected variable location $\delta \in \{1,2,3,\dots D\}$</u>

Only parameter of the algorithm



*δ aims to ensure that at least on variable is obtained from the donor variable

Selection

- Evaluate the fitness of all offspring (f_{u_i})
- Population is updated using greedy selection

Minimization Problem

$$X_i = U_i\,, \qquad if\ f_{u_i} < f_i$$
$$X_i = X_i\,, \qquad if\ f_{u_i} > f_i$$

- Major parent only competes with the corresponding offspring

Other Mutation Strategies

Strategy	Expression for donor vector	Minimum N_p
DE/rand/1	$V = X_{r_1} + F\left(X_{r_2} - X_{r_3}\right)$	4
DE/best/1	$V = \boxed{X_{best}} + F\left(X_{r_1} - X_{r_2}\right)$	3
DE/rand/2	$V = X_{r_1} + F\left(X_{r_2} - X_{r_3}\right) + F\left(\boxed{X_{r_4} - X_{r_5}}\right)$	6
DE/best/2	$V = \boxed{X_{best}} + F\left(X_{r_1} - X_{r_2}\right) + F\left(\boxed{X_{r_3} - X_{r_4}}\right)$	5
DE/target-to-best/1	$V = X_i + F\left(X_{best} - X_i\right) + F\left(X_{r_1} - X_{r_2}\right)$	3