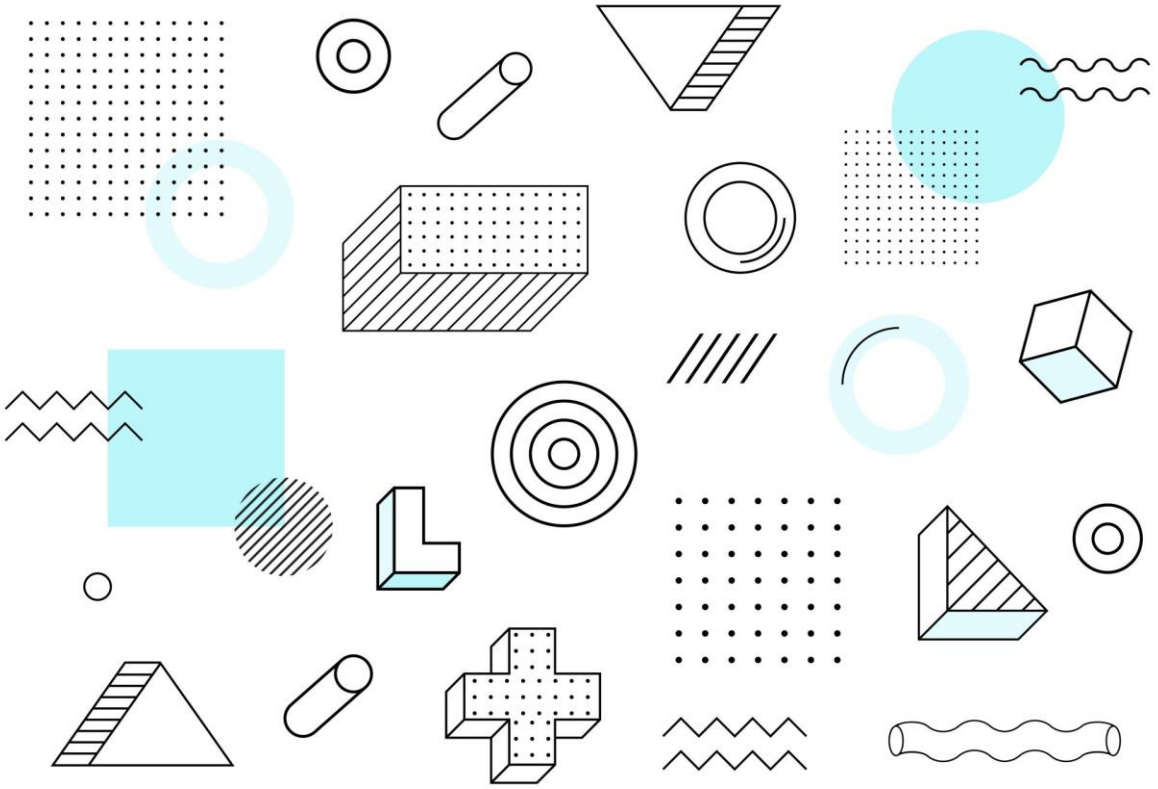
كلية الحاسبات والذكاء الاصطناعي
FACULTY OF COMPUTERS & ARTIFICIAL INTELLIGENCE
BENHA UNIVERSITY



Planning Techniques for Robotics
Lab 07 – Artificial Bee Colony (Practical)

Eng. Yousef Elbaroudy

2025/2026

GUIDELINES

- You don’t need to memorize what will be mentioned. Instead, try to understand

- Each PowerPoint Presentation will be available as PDF file on Google Drive Folder of the Course

Artificial Bee Colony (Recap)

- **Artificial Bee Colony (ABC):** a **nature-inspired optimization algorithm** based on how real honeybees search for and exploit food sources. It's commonly used to solve **continuous and numerical optimization problems**, where the goal is to *find the best solution among many possibilities*.

- In other words, **Artificial Bee Colony** is a way to solve optimization problems based on how bees explore different solutions and share information to gradually focus on the most promising ones.

Motivation

The motivation behind **Artificial Bee Colony (ABC)** was to design a simple and flexible optimization algorithm that can effectively **balance exploration and exploitation**, **avoid local optima, and handle complex non-linear problems**. To achieve this, ABC relies on a population of candidate solutions (food sources) that are continuously improved through local search and replaced by new random solutions, allowing the algorithm to efficiently search for optimal solutions **without complex control mechanisms**.

Artificial Bee Colony (Recap)

- **Artificial Bee Colony (ABC):** a **nature-inspired optimization algorithm** based on how real honeybees search for and exploit food sources. It's commonly used to solve **continuous and numerical optimization problems**, where the goal is to *find the best solution among many possibilities*.

- In other words, **Artificial Bee Colony** is a way to solve optimization problems based on how bees explore different solutions and share information to gradually focus on the most promising ones.

- **Artificial Bee Colony (ABC)** is a **population-based, swarm intelligent, and sign-based algorithm** because it works with a group of agents (bees) that ***collaboratively*** explore and exploit the search space, where solutions are improved through ***collective interaction***, information sharing, and adaptive role-based behaviors rather than refining a single solution independently.

Three types of Worker Bees (Recap)

Employed Bees

Employed bees search for nectar/food sources information from surrounding. and share with the followers.

- ❑ Generate a new solution
- ❑ Calculate new fitness
- ❑ Apply Greedy Selection

Onlooker Bees

Onlooker bees select the bee with better nectar information.

- ❑ Calculate the probabilities
- ❑ Produce a new solution depending on probability
- ❑ Calculate new fitness
- ❑ Apply Greedy Selection

Scout Bees

Scout bees abandon nectar and search for new sources of nectar.

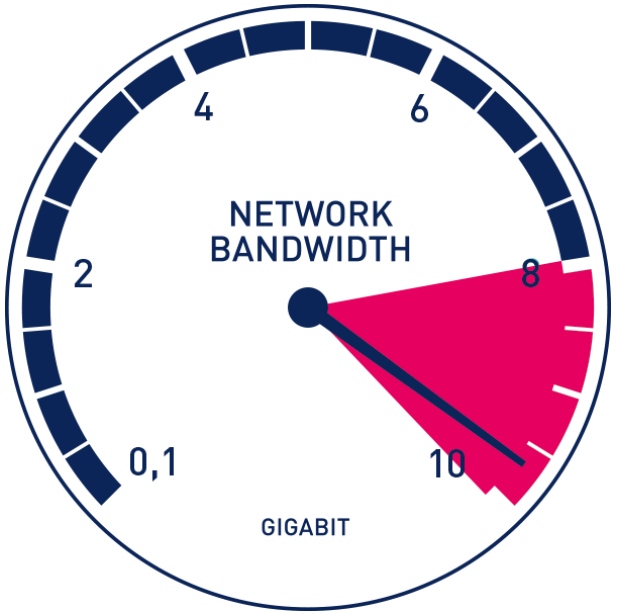
- ❑ Find the abandoned solution (based on variable *limit*)
- ❑ Generate a new solution randomly to replace them

Nectar/Food source is a candidate solution that associated with each employee bee. Its quality is determined by Fitness Value.

Common Problem to Solve

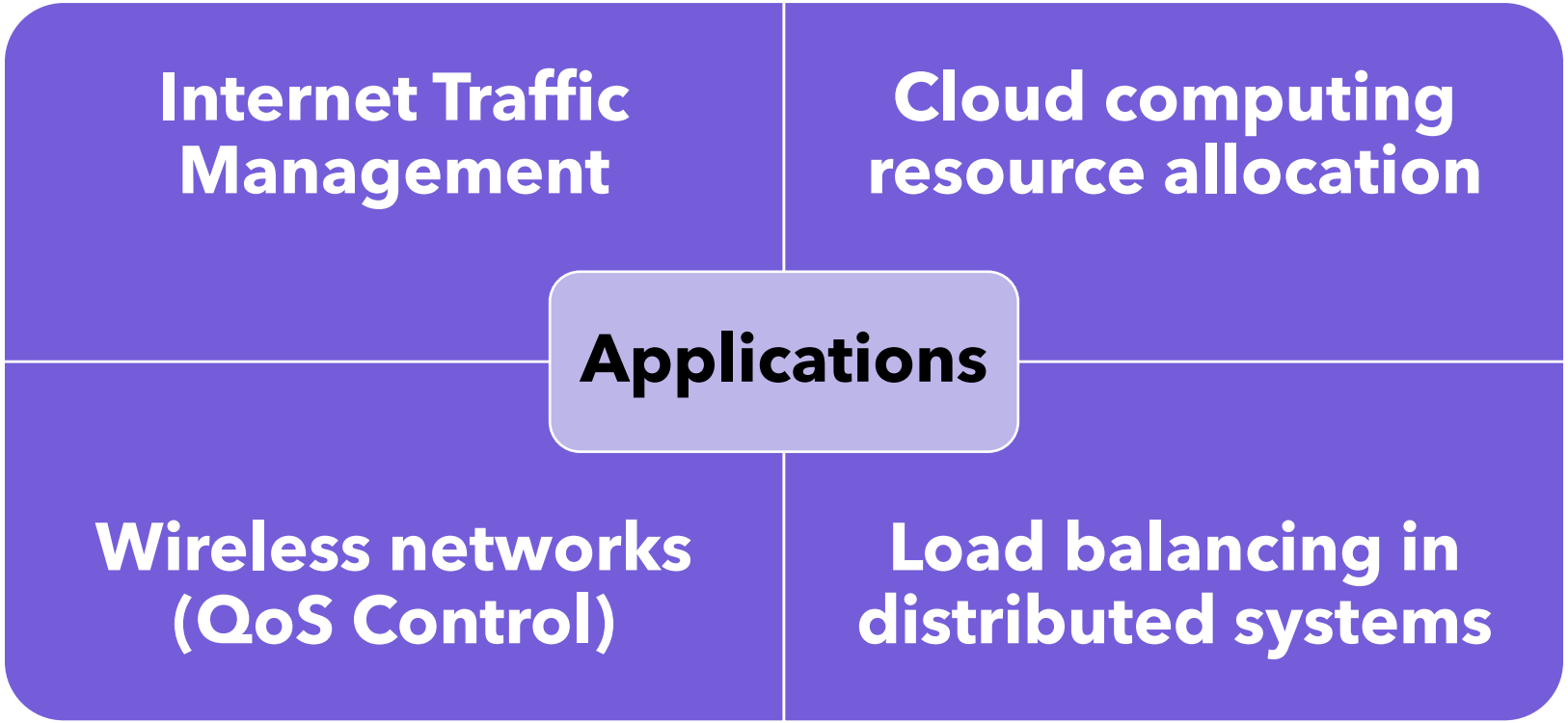
Network Bandwidth Allocation Problem

The **Network Bandwidth Allocation Problem** is a **constrained, non-linear and network optimization problem** where a limited communication bandwidth must be distributed among multiple users, applications, or services in a network in an efficient and **fair manner**.



Network Bandwidth Allocation Problem

Key Idea: Users with **HIGHER PRIORITY** receive proportionally larger bandwidth, but all users receive a share of the resource depending on the optimization objective.



Implementation

Network Bandwidth Allocation Problem

Example to implement

You started as Internet Service Provider (ISP) with bandwidth = 200 Mbps
You are offering 3 tiers of subscriptions as:
Gold Tier (3) - Silver Tier (2) - Bronze Tier (1)

Current Subscribers

Gold Subscriber | Bronze Subscriber | Silver Subscriber | Gold Subscriber

How would you allocate bandwidth FAIRLY ?

Obtain Objective Function

User Priority

Gold Subscriber
Higher

Bronze Subscriber
Lower

Silver Subscriber
Mid

Gold Subscriber
Higher

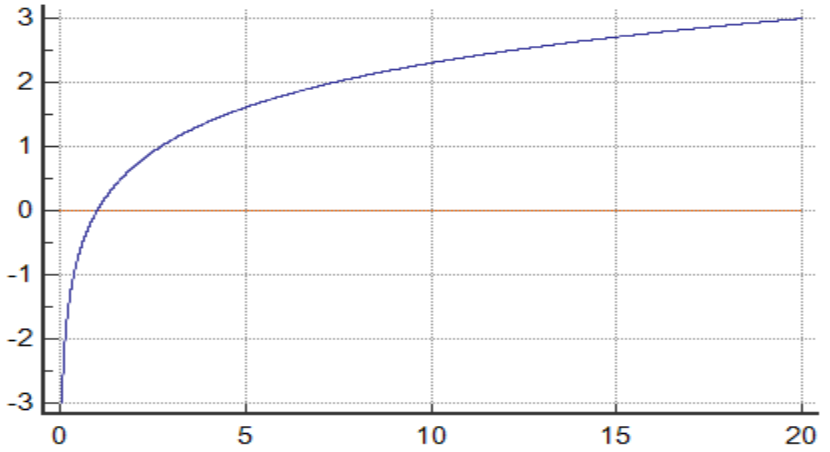
Priorities: 3 1 2 3

Higher priority receive more share bandwidth than the other

Obtain Objective Function (Cont.)

User Satisfaction ☺

A user is satisfied rapidly at some rate of bandwidth. After then, the increasing of bandwidth makes no difference, so the satisfaction is increasing in slower rate



Can be modeled using logarithm → $\ln(bandwidth_i)$

Obtain Objective Function (Cont.)

Priority + Satisfaction

A higher priority user should be more satisfied than the other
So, the total satisfaction among all users can be measured based on their priorities.

Can be modeled using:

$$\text{Total Satisfaction} = \underbrace{\text{Priority}_1 \ln(bw_1)}_{\text{User}_1} + \underbrace{\text{Priority}_2 \ln(bw_2)}_{\text{User}_2} + \ldots + \underbrace{\text{Priority}_n \ln(bw_n)}_{\text{User}_n}$$

In our Problem, Maximize Satisfaction !

$$f(x) = w_1 \ln(x_1) + w_2 \ln(x_2) + w_3 \ln(x_3) + w_4 \ln(x_4)$$

Notes

⚠ Constraints

Each user must receive an amount of bandwidth

So, make sure that $x_i \geq 0.1$ to make sure the user is establishing connection

Max Bandwidth you have is 200

So, a one user cannot have a bandwidth more than 200 $x_i \leq 200$

Fairness among all users of shared bandwidth

So, make sure that $x_1 + x_2 + x_3 + x_4 \leq 200$

Notes

⚠ Constraints

Each user must receive an amount of bandwidth

So, make sure that $x_i \geq 0.1$ to make sure the user is establishing connection

Max Bandwidth you have is 200

So, a one user cannot have a bandwidth more than 200 $x_i \leq 200$

Fairness among all users of shared bandwidth

So, make sure that $x_1 + x_2 + x_3 + x_4 \leq 200$

Any solution that <u>VIOLATES</u> that will be penalized !

Importing Libraries

```
import numpy as np
import random
import math
```

Networking Class (Initialize Parameters)

```
# Represent Network Bandwidth Allocation Problem
class networking():
    def __init__(self, users_priority, no_of_bees, iterations, limit, penalty = None, lb = None, ub = None):
        # Initialize all parameters of the problem
        self.__users_priority = users_priority                    # Priority of users (Subscriptions - Tiers)
        self.__lb = lb                                            # minimum bandwidth a user can receive
        self.__ub = ub                                            # maximum bandwidth a user can receive
        self.__pen = penalty                                      # a penalty of a solution if it exceeded the max bandwidth
        self.__initialize_parameters(no_of_bees, iterations, limit)
```

In constructor, problem parameters are defined

Networking Class (Initialize Parameters)

```python
# Represent Network Bandwidth Allocation Problem
class networking():
    def __init__(self, users_priority, no_of_bees, iterations, limit, penalty = None, lb = None, ub = None):
        # Initialize all parameters of the problem
        self.__users_priority = users_priority                    # Priority of users (Subscriptions - Tiers)
        self.__lb = lb                                            # minimum bandwidth a user can receive
        self.__ub = ub                                            # maximum bandwidth a user can receive
        self.__pen = penalty                                      # a penalty of a solution if it exceeded the max bandwidth
        self.__initialize_parameters(no_of_bees, iterations, limit)
```

Networking Class (Initialize Parameters Method)

```
def __initialize_parameters(self, no_of_bees, iterations, limit):
    # To initialize the hyperparamters of the algorithm
    self.__no_of_bees = no_of_bees
    self.__iterations = iterations
    self.__limit = limit

    self.__dimensions = len(self.__users_priority)     # The dimensions of the problem = number of users
    self.__dims = range(self.__dimensions)             # A list of dimensions [0, 1, 2, ... , n]
```

All Hyperparameters of ABC are initialized

```
# Remember, the form of the solution before proceeding is [population][f(x)][fit][trial]

if self.__lb is not None and self.__ub is not None:
    # If the problem has a boundaries as a constraint
    # X = LowerBound + rand (UpperBound - LowerBound)
    self.__population = np.random.uniform(self.__lb, self.__ub, (self.__no_of_bees, self.__dimensions))
else:
    # If boundaries is not constrained, start with a population of small numbers
    self.__population = np.random.rand(self.__no_of_bees, self.__dimensions)

# Evaluate the objective function of the initialized food sources/nectar
self.__objective_vec = np.array(list(map(lambda x: self.__objective_function(x), self.__population)))

# Check if a solution exceeded the constraint to apply penalty
penalties = np.array(list(map(lambda x: self.__penalty(x), self.__population)))

# Evaluate the fitness function for each solution based on f(x)
self.__fitness_vec = np.array(list(map(lambda x: self.__fitness_function(x), zip(self.__objective_vec, penalties))))

# Set trial of all solutions to zero
self.__trials = np.zeros(self.__no_of_bees)
```

Networking Class (Initialize Parameters Method)

```python
def __initialize_parameters(self, no_of_bees, iterations, limit):
    # To initialize the hyperparamters of the algorithm
    self.__no_of_bees = no_of_bees
    self.__iterations = iterations
    self.__limit = limit

    self.__dimensions = len(self.__users_priority)     # The dimensions of the problem = number of users
    self.__dims = range(self.__dimensions)             # A list of dimensions [0, 1, 2, ... , n]
```

All Hyperparameters of ABC are initialized

Population size = bees x dims

```python
# Remember, the form of the solution before proceeding is [population][f(x)][fit][trial]

if self.__lb is not None and self.__ub is not None:
    # If the problem has a boundaries as a constraint
    # X = LowerBound + rand (UpperBound - LowerBound)
    self.__population = np.random.uniform(self.__lb, self.__ub, (self.__no_of_bees, self.__dimensions))
else:
    # If boundaries is not constrained, start with a population of small numbers
    self.__population = np.random.rand(self.__no_of_bees, self.__dimensions)

# Evaluate the objective function of the initialized food sources/nectar
self.__objective_vec = np.array(list(map(lambda x: self.__objective_function(x), self.__population)))

# Check if a solution exceeded the constraint to apply penalty
penalties = np.array(list(map(lambda x: self.__penalty(x), self.__population)))

# Evaluate the fitness function for each solution based on f(x)
self.__fitness_vec = np.array(list(map(lambda x: self.__fitness_function(x), zip(self.__objective_vec, penalties))))

# Set trial of all solutions to zero
self.__trials = np.zeros(self.__no_of_bees)
```

Networking Class (Objective Function Method)

```python
def __objective_function(self, var):
    # The objective function is MEASURING THE TOTAL SATISFACTION OF USERS BASED ON BANDWIDTH
    # f(x) = w1 x ln(x1) + w2 x ln(x2) + ... + wn x ln(xn) , where w is the priority | xi is the bandwidth for ith user
    obj = [self.__users_priority[x] * math.log(var[x]) for x in self.__dims]
    return sum(obj)
```

In our Problem, Maximize Satisfaction !

$$f(x) = w_1 \ln(x_1) + w_2 \ln(x_2) + w_3 \ln(x_3) + w_4 \ln(x_4)$$

Networking Class (Objective Function Method)

```python
def __objective_function(self, var):
    # The objective function is MEASURING THE TOTAL SATISFACTION OF USERS BASED ON BANDWIDTH
    # f(x) = w1 x ln(x1) + w2 x ln(x2) + ... + wn x ln(xn) , where w is the priority | xi is the bandwidth for ith user
    obj = [self.__users_priority[x] * math.log(var[x]) for x in self.__dims]
    return sum(obj)
```

As a single food source in ABC would be presented as $[x_1, x_2, \dots, x_n]$
We will iterate over each dimension and apply SUB of the objective function

Networking Class (Objective Function Method)

```
def __objective_function(self, var):
    # The objective function is MEASURING THE TOTAL SATISFACTION OF USERS BASED ON BANDWIDTH
    # f(x) = w1 x ln(x1) + w2 x ln(x2) + ... + wn x ln(xn) , where w is the priority | xi is the bandwidth for ith user
    obj = [self.__users_priority[x] * math.log(var[x]) for x in self.__dims]
    return sum(obj)
```

At the end, we will sum all Sub-Terms to obtain the total satisfaction

Networking Class (Penalty Method)

```
def __penalty(self, var):
    # Check if a single solution (all users) exceeded the max bandwidth or not
    if sum(var) > self.__ub:
        return 1              # Exceeded
    else:
        return -1             # Not exceeded
```

Each user is provided with sub-bandwidth, so the total bandwidth for all users mustn’t exceeded the limit. Otherwise, the solution will be penalized !

Networking Class (Penalty Method)

```
def __penalty(self, var):
    # Check if a single solution (all users) exceeded the max bandwidth or not
    if sum(var) > self.__ub:
        return 1              # Exceeded
    else:
        return -1             # Not exceeded
```

Each user is provided with sub-bandwidth, so the total bandwidth for all users mustn't exceeded the limit. Otherwise, the solution will be penalized !

Fairness among all users of shared bandwidth

So, make sure that $x_1 + x_2 + x_3 + x_4 \leq 200$

Any solution that VIOLATES that will be penalized !

Networking Class (Fitness Function Method)

```
def __fitness_function(self, obj_fun):
    # Since it is MAXIMIZATION Problem, we will use the fitness formula of maximization
    # Parameter "obj_fun" --> it is a tuple (fx , penalty)
    if obj_fun[0] >= 0:
        if obj_fun[1] == 1:                      # Solution is penalized or not
            return 1/(obj_fun[0] + self.__pen)  # Penalized solutions will has very low fitness -Closer to Zero- to not be used
        else:
            return obj_fun[0]
    else:
        return 1/(1+abs(obj_fun[0]))
```

In this method, a parameter "*obj_fun*" is provided as TUPLE

It provides f(x) of a solution and if it is penalized or not

Networking Class (Fitness Function Method)

```python
def __fitness_function(self, obj_fun):
    # Since it is MAXIMIZATION Problem, we will use the fitness formula of maximization
    # Parameter "obj_fun" --> it is a tuple (fx , penalty)
    if obj_fun[0] >= 0:
        if obj_fun[1] == 1:                     # Solution is penalized or not
            return 1/(obj_fun[0] + self.__pen)  # Penalized solutions will has very low fitness -Closer to Zero- to not be used
        else:
            return obj_fun[0]
    else:
        return 1/(1+abs(obj_fun[0]))
```

$$fit(I) = \begin{cases} f(x) & , f(x) \geq 0 \\ \frac{1}{1+f(x)} & , f(x) < 0 \end{cases}$$

For maximization problems

Networking Class (Fitness Function Method)

```python
def __fitness_function(self, obj_fun):
    # Since it is MAXIMIZATION Problem, we will use the fitness formula of maximization
    # Parameter "obj_fun" --> it is a tuple (fx , penalty)
    if obj_fun[0] >= 0:
        if obj_fun[1] == 1:                      # Solution is penalized or not
            return 1/(obj_fun[0] + self.__pen)  # Penalized solutions will has very low fitness -Closer to Zero- to not be used
        else:
            return obj_fun[0]
    else:
        return 1/(1+abs(obj_fun[0]))
```

$$fit(I) = \begin{cases} f(x) & , f(x) \geq 0 \\ \frac{1}{1+f(x)} & , f(x) < 0 \end{cases}$$

For maximization problems

Networking Class (Fitness Function Method)

```
def __fitness_function(self, obj_fun):
    # Since it is MAXIMIZATION Problem, we will use the fitness formula of maximization
    # Parameter "obj_fun" --> it is a tuple (fx , penalty)
    if obj_fun[0] >= 0:
        if obj_fun[1] == 1:                          # Solution is penalized or not
            return 1/(obj_fun[0] + self.__pen)       # Penalized solutions will has very low fitness -Closer to Zero- to not be used
        else:
            return obj_fun[0]
    else:
        return 1/(1+abs(obj_fun[0]))
```

A penalized solution's (exceeded maximum bandwidth) fitness will be computed as $\frac{1}{f(x)+penalty}$ to make it with lower fitness (closer to 0)

Networking Class (Greedy Selection Method)

```
def __greedy_selection(self, solutions, objectives, fitness, trial):
    # Parameters: pair of elements, the first element for the current solution, and second element for the new generated solution
    # Remember, the form is [food sources][f(x)][fit][trial]
    # Compare between new solution and current solution based on FITNESS

    if fitness[1] > fitness[0]: # New solution is better or not
        return [solutions[1], objectives[1], fitness[1], 0]
    else:
        return [solutions[0], objectives[0], fitness[0], trial + 1]
```

In this method, all parameters are introduced as TUPLES as contain an element for current solution (index 0) and new solution (index 1)

Networking Class (New Solution Method)

```
def __new_solution(self, current_solution, random_partner, random_variable):
    # The procedure of creating new solution
    # Xnew = X + Φ (X - Xp)
    # Evaluate new solution f(x) and fitness

    phi = np.random.uniform(-1,1)
    X = current_solution[random_variable]
    Xp = random_partner[random_variable]
    Xnew = X + phi * (X - Xp)

    # If boundaries are constrained, Xnew will be bounded
    if self.__lb is not None and Xnew < self.__lb:
        Xnew = self.__lb
    elif self.__ub is not None and Xnew > self.__ub:
        Xnew = self.__ub

    new_solution = current_solution.copy()
    new_solution[random_variable] = Xnew
    new_objective_func = self.__objective_function(new_solution)
    penalty = self.__penalty(new_solution)
    new_fitness_func = self.__fitness_function((new_objective_func,penalty))
    return new_solution, new_objective_func, new_fitness_func
```

This method is common for Employed Phase and Onlooker Phase. So, it is separated in a standalone method

Networking Class (Employed Phase Method)

```python
def __employed_phase(self):
    # All employed bees are encountering all food sources existed
    # Offline update is required here (No inplace modification)
    population = []
    objective_vec = []
    fitness_vec = []
    trials = []

    for ix in range(self.__no_of_bees): # Iterate over all employed bees
        current_solution = self.__population[ix]
        current_obj = self.__objective_vec[ix]
        current_fit = self.__fitness_vec[ix]

        random_partner = random.choice(self.__population) # Pick a random food source
        random_variable = np.random.choice(self.__dims)   # Pick a random variable to modify
        # Generate the new solution
        new_solution, new_objective_func, new_fitness_func = self.__new_solution(current_solution, random_partner, random_variable)

        # Apply Greedy Selection (Comparison between New and current solution)
        selected_sol, selected_obj, selected_fit, trial = self.__greedy_selection((current_solution, new_solution),
                                                                                  (current_obj, new_objective_func),
                                                                                  (current_fit, new_fitness_func), self.__trials[ix])

        population.append(selected_sol)
        objective_vec.append(selected_obj)
        fitness_vec.append(selected_fit)
        trials.append(trial)

    # After Iterating over all bees, update requirements (Offline update)
    self.__population = population
    self.__objective_vec = objective_vec
    self.__fitness_vec = fitness_vec
    self.__trials = trials
```

Networking Class (Onlooker Phase Method)

Onlooker Bees

Onlooker bees select the bee with better nectar information.

- ❑ Calculate the probabilities
- ❑ Produce a new solution depending on probability
- ❑ Calculate new fitness
- ❑ Apply Greedy Selection

```
def __onlooker_phase(self):
    # All onlooker bees are encountering food sources based on their PROBABILITIES
    probabilities = self.__fitness_vec.copy()
    sum_of_fit = sum(probabilities)
    probabilities = np.array(probabilities) / sum_of_fit

    ix = 0
    for _ in range(self.__no_of_bees):
        while True: # Circular Loop to iterate over all food sources BASED ON THEIR PROBABILITIES
            random_number = np.random.random()
            if random_number <= probabilities[ix]:
                current_solution = self.__population[ix]
                current_obj = self.__objective_vec[ix]
                current_fit = self.__fitness_vec[ix]

                random_partner = random.choice(self.__population)
                random_variable = np.random.choice(self.__dims)
```

```
    new_solution, new_objective_func, new_fitness_func = self.__new_solution(current_solution, random_partner, random_variable)

    selected_sol, selected_obj, selected_fit, trial = self.__greedy_selection((current_solution, new_solution),
                                                                              (current_obj, new_objective_func),
                                                                              (current_fit, new_fitness_func), self.__trials[ix])

    # Apply ONLINE UPDATE
    self.__population[ix] = selected_sol
    self.__objective_vec[ix] = selected_obj
    self.__fitness_vec[ix] = selected_fit
    self.__trials[ix] = trial
    ix = (ix + 1) % self.__no_of_bees
    break
ix = (ix + 1) % self.__no_of_bees
```

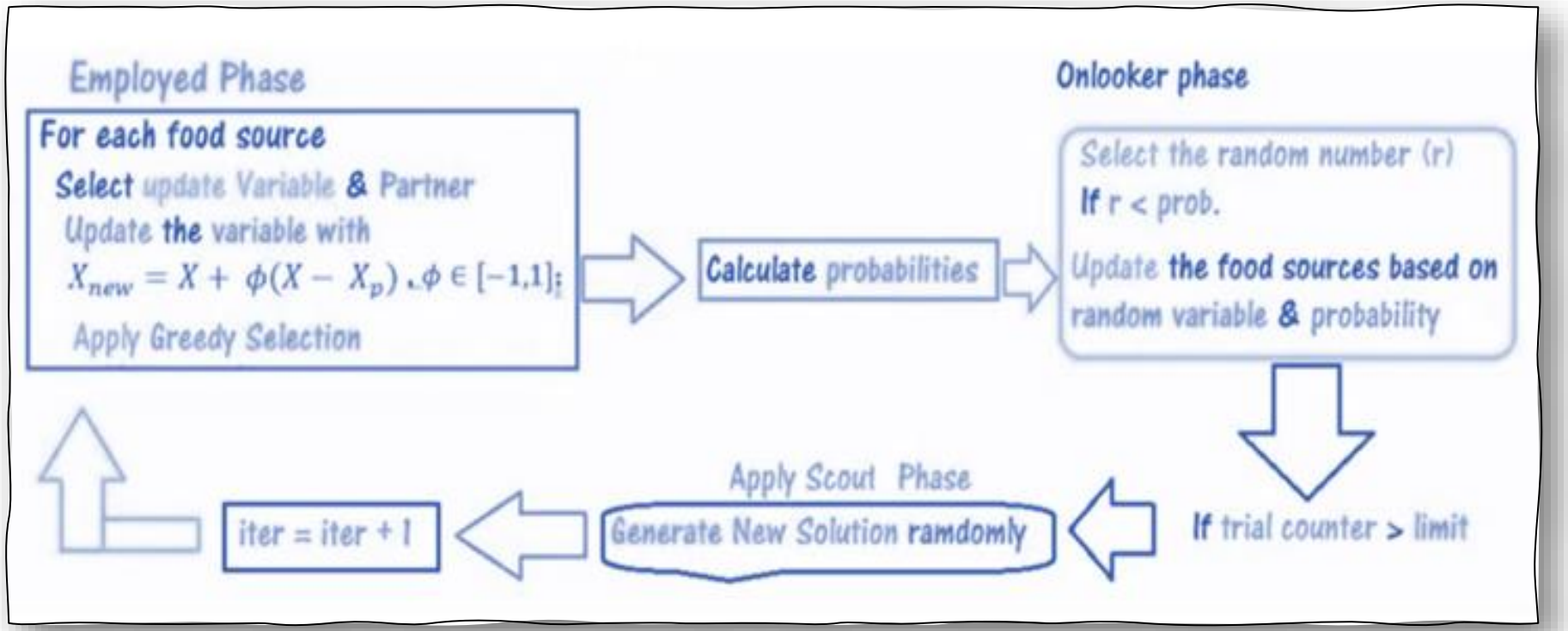
Networking Class (Scout Phase Method)

```python
def __scout_phase(self):
    # Check if a food source/nectar is fully utilized -Its trial exceeded limit-
    for ix in range(self.__no_of_bees):
        if self.__trials[ix] > self.__limit:
            if self.__lb is not None and self.__ub is not None:
                self.__population[ix] = np.random.uniform(self.__lb, self.__ub, (self.__dimensions))
            else:
                self.__population[ix] = np.random.rand(self.__dimensions)
            self.__objective_vec[ix] = self.__objective_function(self.__population[ix])
            penalty = self.__penalty(self.__population[ix])
            self.__fitness_vec[ix] = self.__fitness_function((self.__objective_vec[ix],penalty))
            self.__trials[ix] = 0
```

Scout Bees

Scout bees abandon nectar and search for new sources of nectar.

- ❑ Find the abandoned solution (based on variable *limit*)
- ❑ Generate a new solution randomly to replace them



Artificial Bee Colony Steps

Networking Class (Solution Method)

```
def solution(self):
    best_solution_ever = None
    best_solution_obj = None
    best_solution_fit = None
    for iteration in range(self.__iterations):
        self.__employed_phase() # Applying Employed Phase
        self.__onlooker_phase() # Applying Onlooker Phase

        # Elitism: Save the best solution before Scout Phase, to guarantee the best soluton is not removed
        best_index = self.__fitness_vec.index(max(self.__fitness_vec))
        best_solution = self.__population[best_index]
        best_obj = self.__objective_vec[best_index]
        best_fit = self.__fitness_vec[best_index]
        if best_solution_fit is None or best_fit > best_solution_fit:
            best_solution_ever = best_solution
            best_solution_obj = best_obj
            best_solution_fit = best_fit

        self.__scout_phase()   # Applying Scout Phase
        print(f"\rCurrent Iter: {iteration+1:06}, Best solution ever: {np.round(best_solution_ever,2)} with Fitness = {best_solution_fit:.2f}", end="")

    return best_solution_ever,best_solution_obj
```

Networking Class (Solution Method)

```
def solution(self):
    best_solution_ever = None
    best_solution_obj = None
    best_solution_fit = None
    for iteration in range(self.__iterations):
        self.__employed_phase() # Applying Employed Phase
        self.__onlooker_phase() # Applying Onlooker Phase

        # Elitism: Save the best solution before Scout Phase, to guarantee the best soluton is not removed
        best_index = self.__fitness_vec.index(max(self.__fitness_vec))
        best_solution = self.__population[best_index]
        best_obj = self.__objective_vec[best_index]
        best_fit = self.__fitness_vec[best_index]
        if best_solution_fit is None or best_fit > best_solution_fit:
            best_solution_ever = best_solution
            best_solution_obj = best_obj
            best_solution_fit = best_fit

        self.__scout_phase()    # Applying Scout Phase
        print(f"\rCurrent Iter: {iteration+1:06}, Best solution ever: {np.round(best_solution_ever,2)} with Fitness = {best_solution_fit:.2f}", end="")

    return best_solution_ever,best_solution_obj
```

Elitism: Keep the best solution surviving for later iterations

Testing

```
problem = networking(users_priority=[3,1,2,3], no_of_bees=25, iterations=30000, limit=30, penalty=10000, lb=0.1, ub=200)
sol, obj = problem.solution()
```

```
Current Iter: 030000, Best solution ever: [63.81 23.7  45.46 66.99] with Fitness = 35.88
```

Let's see if it is true answer ?

Proportional Fairness
The total bandwidth is proportionate to priority

$$x_i = B \times \frac{w_i}{\sum w}$$

Our solution:
[63.81 , 23.70 , 45.46 , 66.99]

Let's see if it is true answer ?

Proportional Fairness
The total bandwidth is proportionate to priority

$$x_i = B \times \frac{w_i}{\sum w}$$

User	Shared Bandwidth for user (x_i)
1	66.67 Mbps
2	22.22 Mbps
3	44.44 Mbps
4	66.67 Mbps

Our solution:
[63.81 , 23.70 , 45.46 , 66.99]

Let's see if it is true answer ?

Proportional Fairness
The total bandwidth is proportionate to priority

$$x_i = B \times \frac{w_i}{\sum w}$$

User	Shared Bandwidth for user (x_i)
1	66.67 Mbps
2	22.22 Mbps
3	44.44 Mbps
4	66.67 Mbps

Our solution:
[63.81 , 23.70 , 45.46 ,]

Near-Optimal Solution

Behavior of Bees and Trials

```
([array([1.91627284e+01, 1.40782990e+02, 7.22686360e+01, 1.00000000e-01]), [4,
  array([1.57329544e+02, 9.49079049e+01, 8.89952445e+01, 1.00000000e-01]),  4,
  array([1.49078046e+02, 1.00000000e-01, 1.61391555e+01, 1.35282035e+02]),  1,
  array([102.57587789, 108.5957631 ,  52.41092868, 131.68738787]),          0,
  array([17.22916091, 55.50597447, 78.19296499, 62.84998287]),              3,
  array([ 53.29487725,   7.4474814 , 113.66736982,  25.11398358]),          10,
  array([ 20.55504719,  32.05997095,  15.80464594, 114.58555721]),          1,
  array([1.00000000e-01, 7.30671235e+01, 1.69487244e+02, 7.13220692e+01]),  3,
  array([64.80153672, 27.44126053, 32.89002048, 47.31269163]),              0,
  array([80.14717372,  2.29831735, 68.79283179, 47.79784977]),              3,
  array([124.41960993,  18.98853561,  25.00129942,  30.73539525]),          15,
  array([  6.9508106 , 165.30004463,  20.66103872,   6.40819633]),          4,
  array([119.51151521,   2.5386977 ,  43.47925455,  34.3310809 ]),          6,
  array([1.00000000e-01, 1.78289889e+02, 7.46381704e+01, 1.00000000e-01]),  8,
  array([107.55164246,  16.59019965, 191.97637791,  28.34349062]),          3,
  array([  9.23162975,  44.30908495, 145.09780097,   1.35403162]),          30,
  array([7.19138299e+00, 1.17993172e+02, 7.46645855e+01, 1.08271340e-01]),  14,
  array([103.42858469,  96.10029453,  12.72997528, 105.55451083]),          4,
  array([ 27.60391955, 126.08236379,   8.01738466,  31.1873974 ]),          2,
  array([34.67797355, 50.18173007, 39.22500073, 75.5823474 ]),              4,
  array([179.17911613,  17.40601694,  62.36711213, 138.01621602]),          0,
  array([ 13.67792552, 117.81102405,  38.18468301,  29.6460361 ]),          0,
  array([1.33435482e+02, 1.00114428e+00, 2.53208644e+01, 1.00000000e-01]),  0,
  array([146.11398167,  16.94001599,  28.30359317,  34.78208432]),          2,
  array([176.13314674,  95.32110086,  13.34147031,  77.08358157])],         0])
```

Thank You

Next Lab:
Particle Swarm Optimization (Theoretical)